2013

Testing and Validation of a Prototype Gpgpu Design for FPGAs

Murtaza Merchant
University of Massachusetts Amherst

TESTING AND VALIDATION OF A PROTOTYPE GPGPU DESIGN FOR FPGAs

A Thesis Presented

by

MURTAZA S. MERCHANT

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL AND COMPUTER ENGINEERING

February 2013

Department of Electrical and Computer Engineering

TESTING AND VALIDATION OF A PROTOTYPE GPGPU DESIGN FOR FPGAs

A Thesis Presented

by

MURTAZA S. MERCHANT

ACKNOWLEDGEMENTS

To begin with, I would like to sincerely thank my advisor, Prof. Russell Tessier for all his support, faith in my abilities and encouragement throughout my tenure as a graduate student. Without his guidance, this thesis wouldn’t have been possible. I am also very thankful to Kevin Andryc, who has been my constant tutor throughout this project. I couldn’t have asked for a better teammate to work with. His focus and dedication to this project despite a full-time job is something I will always appreciate. I extend my gratitude towards Prof. Wayne Burleson and Prof. Mario Parente, and would like to thank them for being on my thesis committee.

Next, I would like to thank all my wonderful current and former lab mates— Hari, Deepak, Kekai, Cory, Justin, Jia and Gayatri, for making the RCG lab a fun place to work. I will relish the times we spent together for a very long time. I would also like to thank all my other friends that I have made over the past 2 years in Amherst, for making my stay so enjoyable. The town and its people have made my stay a truly worthwhile experience.

And of course, no acknowledgement is complete without expressing your gratitude and thankfulness towards one’s parents. They have always been, and will always be there through my best and worst of times. I deeply thank them for their support and faith in me. I feel truly blessed to have them in my life.

ABSTRACT

TESTING AND VALIDATION OF A PROTOTYPE GPGPU
DESIGN FOR FPGAs

FEBRUARY 2013

MURTAZA S. MERCHANT

B.E, UNIVERSITY OF MUMBAI, INDIA

M.S. E.C.E., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Russell G. Tessier

Due to their suitability for highly parallel and pipelined computation, field programmable gate arrays (FPGAs) and general-purpose graphics processing units (GPGPUs) have emerged as top contenders for hardware acceleration of high-performance computing applications. FPGAs are highly specialized devices that can be customized to an application, whereas GPGPUs are made of a fixed array of multiprocessors with a rigid architectural model. To alleviate this rigidity as well as to combine some other benefits of the two platforms, it is desirable to explore the implementation of a flexible GPGPU (soft GPGPU) using the reconfigurable fabric found in an FPGA. This thesis describes an aggressive effort to test and validate a prototype GPGPU design targeted to a Virtex-6 FPGA. Individual stages of the design have been separately tested with the aid of manually-generated register transfer level (RTL) testbenches and logic simulation tools. The tested modules are then integrated together to build the GPGPU processing pipeline. The GPGPU design is completely validated by benchmarking the platform against five standard CUDA benchmarks with varying control-flow characteristics. The architecture is fully CUDA-compatible and supports direct CUDA compilation of the benchmarks to a binary that is executable on the soft

GPGPU. The validation is performed by comparing the FPGA simulation results against the golden references generated using corresponding C/C++ executions. The efficiency and scalability of the soft GPGPU platform is validated by varying the number of processing cores and examining its effect on the performance and area. Preliminary results show that the validated GPGPU platform with 32 cores can offer up to 25x speedup for most benchmarks over a fully optimized MicroBlaze soft microprocessor. The results also accentuate the benefits of the thread-based execution model of GPUs as well as their ability to perform complex control flow operations in hardware. The testing and validation of the designed soft GPGPU system, serves as a prerequisite for rapid design exploration of the platform in the future.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

In recent years, general purpose computing using graphics processing units (GPUs) has drawn considerable interest in the field of high-performance computation. With many-core processor architecture and a highly parallel programming model, GPUs have multifold computational capabilities for parallel data applications as compared to a modern day CPU (Figure 1).

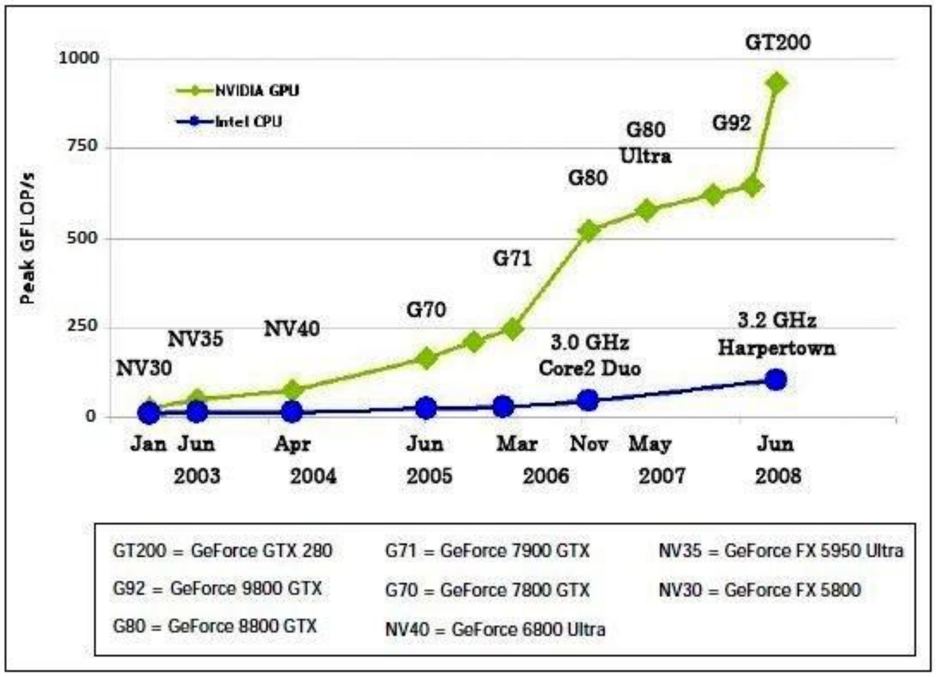


Figure 1: Computations per second - GPU vs. CPU

The advent of high-level programming models like Nvidia's Compute Unified Device Architecture (CUDA) and ATI Stream technology have helped isolate developers from low-level hardware details. However, a limitation of GPGPUs is their rigid architectural model, which is constrained to fixed microarchitectural templates. As a result, there are many computing systems which do not contain a GPGPU. Conversely, field programmable gate arrays (FPGAs) are highly specialized devices that offer application-specific customizations to the designer [1][2]—unfortunately, these optimizations require cumbersome hardware design language (HDL) coding, hardware skills and techniques, beyond the expertise of many developers. Between the hand-modeled FPGA solutions and the high-level programming based GPGPUs, there is sizable design space that warranties systematic exploration. This is depicted by the brown area in Figure 2.

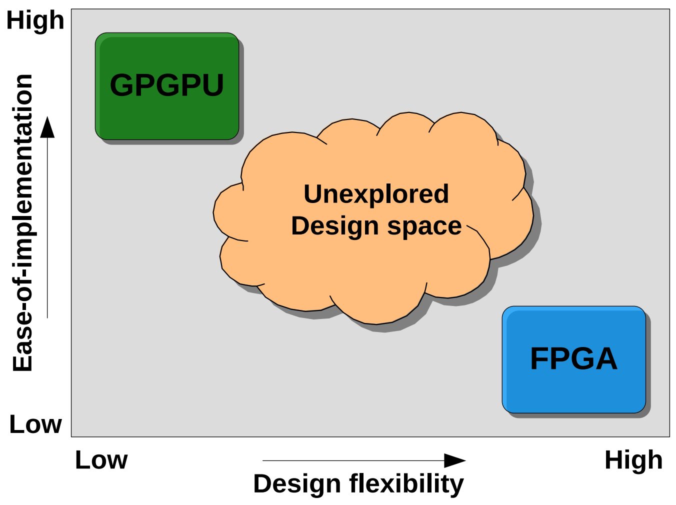


Figure 2: Ease-of-implementation vs. Design Flexibility for GPGPUs and FPGAs

To exploit the respective strengths of these two platforms while simultaneously alleviating their drawbacks, this thesis explores the testing and validation of a prototype GPGPU design targeted to FPGAs, also known as a soft GPGPU. The soft GPGPU is based on the G80 architecture [3]—the first dedicated general-purpose GPU from Nvidia with compute capability 1.0 [4]. Most of its key features like multithreading, vector processing and hardware conditional execution is retained in our FPGA implementation.

An FPGA provides GPGPU flexibility for systems which do not contain an available GPGPU. This flexibility can be expressed in terms of architectural parameters such as the number of processing cores, the arithmetic bit-width, and the ratio of arithmetic elements to memory elements, etc. Our approach enables the reusability of existing system FPGAs for computing applications like image processing and computer vision algorithms that benefit from a many-core architectural template.

The first prototype design of the soft GPGPU has been developed in conjunction with UMass ECE Ph.D student Kevin Andryc. The functional verification of the individual soft GPGPU blocks and the integrated design is necessary before detailed experimentation and evaluation can be carried out. As with any prototype design, verification and validation plays a critical role in the development process. It enables the detection of errors and allows for bug correction early in the design cycle, especially during the implementation of the RTL design from a behavioral specification. As the design process matures, iterative verification is crucial in moving the design forward to the next stage. In this context, a testing and validation plan for the soft GPGPU along with preliminary experimentation results are detailed in this thesis.

There are primarily two contemporary verification techniques that are used in the industry, formal verification [5] [6] and simulation-based verification. Formal verification methods, like equivalence checking, model checking, and theorem-proving, use abstract mathematical models to prove or disprove the correctness of a design. However, using formal verification methods demands experienced designers, who are knowledgeable about various design practices. Today, the industry is more reliant on simulation-based verification techniques, which involve predicting the functional response of a circuit based on specified input values. Logic simulation has been the workhorse verification technique for testing RTL designs. The input values which form the ‘testbench’ are manually created using hardware description languages like VHDL or Verilog and serve as the stimulus to the design. The testbench and the design are fed to logic simulation tools to verify the correctness of the design by comparing the captured simulation waveforms with the expected results based on the specifications of the design.

The testing and validation of the implemented soft GPGPU is performed using simulation-based verification techniques. We begin with testing the individual stages of the pipeline using manually generated VHDL testbenches and making necessary design modifications (if any) commensurate with the required functionality of the module. With sufficient confidence in the correct functionality of the individual stages, we proceed to integrate the stages together to build the GPGPU processing pipeline. Pipeline verification is carried out by simulating a wide variety of CUDA assembly instructions through the pipeline. As a final step, validation of the entire system is accomplished by compiling five CUDA benchmarks to binary and simulating them on the soft GPGPU design. Further, benchmarking experiments are conducted to analyze the effects of reconfiguring certain preliminary architectural parameters of the soft GPGPU on area and performance.

The thesis focuses on leveraging the first-ever soft GPGPU prototype to successfully simulate CUDA benchmarks. It overlays the foundation for conducting a wide variety of experiments in the future and opens up opportunities to compare our implementation with similar parallel processing platforms like FPGA based soft vector processors and OpenCL to multicore implementations.

The rest of the thesis is organized as follows: Chapter 2 provides general background on the Nvidia G80 architecture and the CUDA programming model. This chapter also provides an overview of the related work which includes several FPGA-targeted projects for implementing data parallel applications. Chapter 3 describes the architecture of the implemented soft GPGPU with detailed functionality of critical blocks in the design. Chapter 4 illustrates the testing and validation methodologies used in this work. Chapter 5 discusses the various experiments and explains the obtained results. Chapter 6 concludes the thesis by providing directions for future work.

CHAPTER 2

BACKGROUND AND RELATED WORK

2.1 Field-programmable gate array (FPGA)

A field-programmable gate array (FPGA) is an integrated circuit that can be completely reconfigured even after it is fabricated [7]. As illustrated by

Figure 3, it consists of a prebuilt array of combinational logic blocks (CLBs), memory elements (Block RAMs), input-output blocks (IOBs) and DSP units, surrounded by programmable routing resources that can be configured using a hardware description language (HDL) such as VHDL or Verilog.

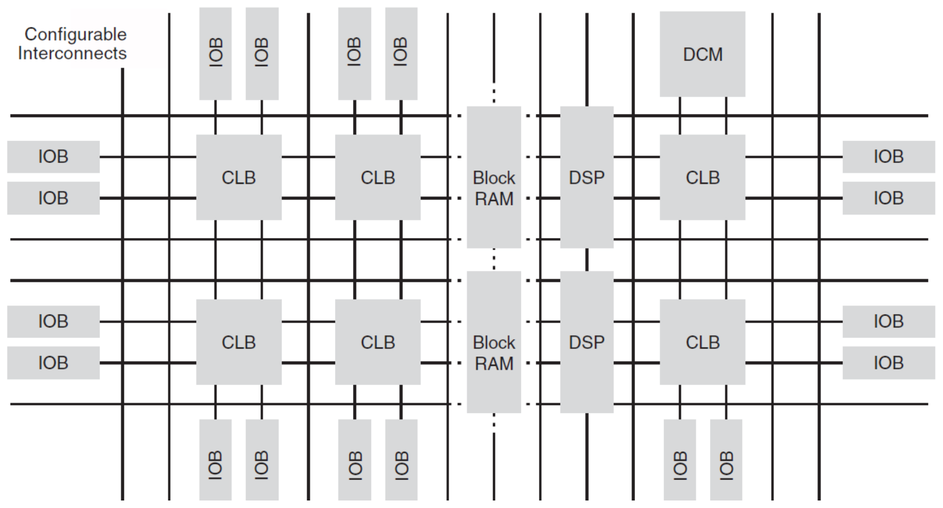


Figure 3: FPGA architecture [8]

Using an HDL, custom hardware functionality can be implemented on an FPGA. The large array of logic blocks spread across the fabric provides fine-grained parallelism to FPGAs. Such parallelism provides orders of magnitudes of application speedup as compared to conventional CPUs, and in some cases even GPUs [9].

FPGAs are highly specialized devices that offer application-specific customization to designers. These customizations include on-chip memory (block RAMs), DSP units (multiplier or a floating point unit), bit-width variations etc. Higher power efficiency [10] and lower time-to-market as compared to ASICs are other benefits provided by FPGAs. These benefits come at the cost of increased design effort and necessity of digital hardware design knowledge for programming these custom-computing machines. However, the advent of high-level synthesis tools [11], with new technologies that convert C code or even graphical descriptions into digital hardware is changing this trend.

2.2 General-purpose computing on graphics processing unit (GPGPU)

General-purpose computing on graphics processing unit (GPGPU) uses graphics processors (GPUs) which typically handle computations for computer graphics and for non-graphics computing applications. GPGPUs have a many-core device architecture and possess substantial parallel processing capabilities [12] [13] [14].They consist of an array of multiprocessors (each with two or more processing units) enabling them to execute thousands of threads in parallel. In a GPU, a majority of the silicon area is dedicated to data processing units with only a small portion assigned to data caching and flow control circuitry, as illustrated by Figure 4. Such a design architecture makes them suitable for solving compute-intensive problems. In comparison, CPUs embrace a sequential data flow structure and are more suited for control flow intensive problems.

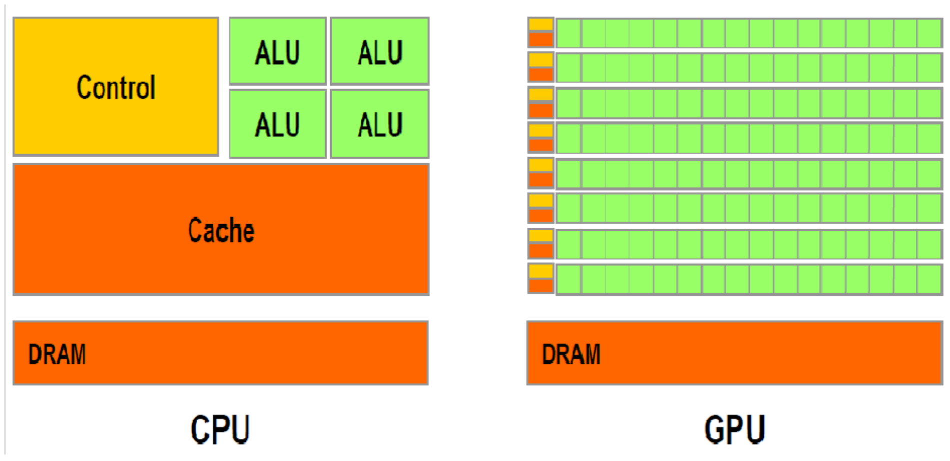


Figure 4: Resource distribution for a CPU and GPU [4]

(Green: Data processing; Yellow: Control; Orange: Memory)

CPUs implement intelligent data caches [4] and flow control mechanisms like dynamic branch prediction [15] to boost performance. Such techniques enable CPUs to hide the latency involved with long memory operations. In contrast, GPUs run thousands of threads in parallel on several processors to execute an application. They rely on dedicated thread scheduling techniques and fast switching between tasks to overshadow memory latency [16]. The primary goal is to achieve maximum multiprocessor occupancy, resulting in high throughput. While one thread is occupied with a long memory operation, other threads can be scheduled in parallel to carry out fast arithmetic operations. The availability of a large number of processors facilitates effective thread scheduling. As most applications targeting GPUs are highly parallel in nature, the abundant processing cores can be exploited to eliminate the need for speculative execution and advanced flow control logic.

A high-level block diagram of the G80 GPU is illustrated in Figure 5. The following sections describe the hardware architecture and the software model of GPUs.

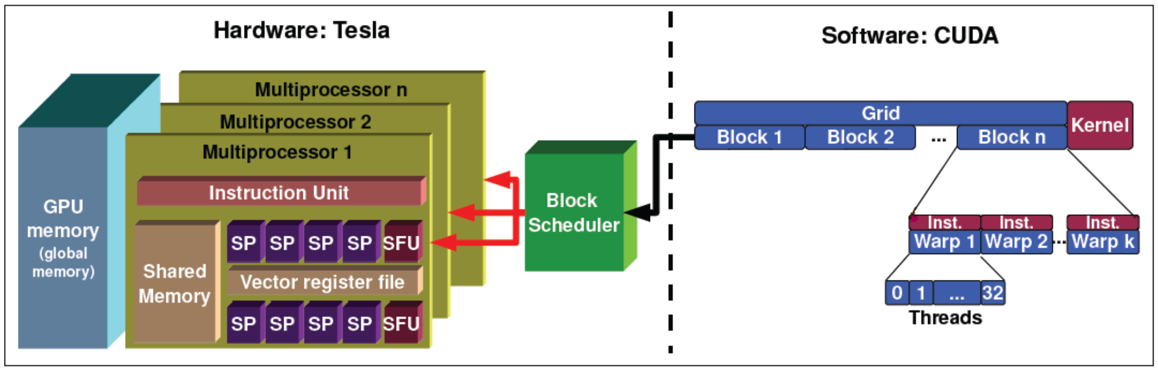


Figure 5: Nvidia GPU environment [17]

2.2.1 G80: The hardware architecture

G80 [3] [18] is Nvidia's first supercomputing processor architecture with dedicated support for general-purpose computing on graphics processors. The high-level architecture of a G80 GPU is shown in Figure 5. The GPU is primarily made up of an array of (streaming) multiprocessors, with each multiprocessor consisting of eight scalar processor (SP) cores. The term '*streaming multiprocessor*' implies that each multiprocessor consists of processing elements that perform the same operation on multiple data simultaneously. This type of execution is termed single instruction, multiple data (SIMD) processing. Each SP operates on a thread, the smallest unit of execution in the GPGPU system. SPs consist of dedicated hardware resources to perform arithmetic and logical operations on threads. The vector register file contains a pool of registers that is strictly partitioned across SPs. The register file striping allows each SP to use its own set of registers for storing operands and results, also steering them away from any data dependent hazards. The shared memory serves as a communication medium between the different SPs residing in the same SM. Communication across different SMs is orchestrated via the global memory, which is accessible to all the threads within the GPU. The global memory is physically implemented as an off-chip DRAM, thus requiring a long memory access time (400-600 clock cycles) [19] as compared to accessing registers or shared memory. In addition, there is a read-only *constant* memory (not shown in the figure) accessible by all the threads. The constant memory space is a cache for each SM, thus allowing fast data access as long as all threads read the same memory address. Special function units (SFUs) perform exclusive arithmetic operations like sine, cosine, logarithmic arithmetic etc. The instruction unit maps program instructions to every thread in the SM.

The number of threads residing in a streaming multiprocessor is governed by the number of registers used per thread. If R is the total number of registers per SM and r is the number of registers per thread, a maximum of R/r threads can be accommodated. The amount of available shared memory also influences the processor occupancy, as it is shared among all the threads of the SM and cannot exceed the available physical resources.

2.2.2 CUDA: The software programming model

In Nvidia architecture, individual CUDA threads are combined together into groups called as *warps,* as shown in Figure 5. Each warp consists of 32 threads which execute the same instruction together in a lockstep fashion. A warp is considered to be the unit for scheduling threads within the SM. When a SM gets a new instruction, it selects a ‘ready’ warp and maps the instruction to every thread within the warp. This process is known as *warp scheduling*. The warp scheduling is critical in masking long latency operations that consume a large number of clock cycles. In case of memory operations, while one warp is busy executing the time consuming *load/store* operations, the SM can schedule another 'ready' warp for execution, thus masking the long memory latency. In this way, the SM manages and executes concurrent threads in hardware with zero scheduling overhead. The zero-overhead thread scheduling enables fine-grained thread-level parallelism in GPUs.

To manage fine-grained thread parallelism, each multiprocessor is architected as a single instruction, multiple-thread (SIMT) processor. As in the SIMD model, every thread performs the same operation on a different set of data and is free to independently execute data-dependent branches. Branching threads diverge from the normal execution flow and hence have to be masked during execution of the non-branching path. As the threads within the warp have to be executed in a lockstep, the instructions pointed to by the branching threads are executed serially, one thread at a time, while the non-branching threads are masked. In case of thread diversion, it is evident that the thread-level parallelism is not fully exploited thus penalizing throughput performance. In the worst case, if there are *n* threads each of which diverging to a different address in a hierarchical fashion, *n* distinct paths would have to be serially executed causing *O (n)* performance penalty.

A thread *block* is formed by combining a fixed number of warps (24 in our case) together. The thread blocks are assigned to different SMs by the *block scheduler*, as shown in Figure 5. A thread block contains threads that can cooperate together and hence it is also called as a cooperative thread array (CTA). Thread synchronization within the same block is achieved by using the __*syncthreads* barrier synchronization instruction.

Threads from different blocks need not synchronize, hence allowing different blocks to execute independently. Due to a limit to the number of threads a block can contain, several thread blocks are combined together to form a *grid* that contains a much larger number of threads. The grid executes the *kernel*—the function to be executed on the GPU. When a kernel is invoked by the host CPU, a grid of threads is launched as an array of parallel thread blocks (CTAs), as shown in Figure 6. Blocks and grids can be one, two or three dimensional, and their size must be specified while launching the kernel.

The hierarchical thread structure defines the compute unified device architecture (CUDA) programming model and directly maps to the GPU hardware architecture as shown in Figure 5. To summarize the hardware-software interaction using a top-down approach:

i. Each block from the grid of threads is assigned to an SM by the block scheduler.
ii. Within each scheduled block, the SM selects idle warps for execution.
iii. Within each warp, each thread is executed on an individual SP.

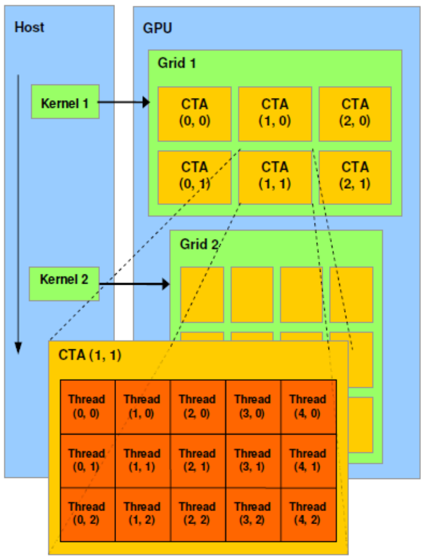


Figure 6: Host-GPU interaction [4]

The CUDA specifications as per compute capability 1.0 are summarized in Table 1. The soft GPGPU architecture is designed based upon these specifications:

Maximum number of resident threads per multiprocessor	256
Maximum number of resident warps per multiprocessor	24
Warp size	32
Maximum number of resident blocks per multiprocessor	3
Maximum number of threads per SM	768
Maximum dimensionality of thread block	3
Maximum dimensionality of a grid of thread block	2
Number of 32-bit registers per multiprocessor	8192
Maximum amount of shared memory per multiprocessor	16 KB
Number of shared memory banks	1
Constant memory size	8 KB

Table 1: CUDA specifications for compute capability 1.0

2.3 Related work / Overall motivation for soft GPGPUs

Over the past few years, FPGA computing using GPGPU microarchitectural templates has been a topic of active research [20][21][22][23]. GPU programming models like Nvidia’s CUDA and AMD’s ATI are gaining traction, and it is less clear if similar programming models defined for FPGAs can be beneficial.

Lebedev et al. [20] to the best of our knowledge were the first to embrace a many-core abstraction for FPGA-based computation. They proposed a many-core approach to a reconfigurable computing (MARC) system for high performance applications expressed in high-level programming languages like OpenCL. The prototype machine was implemented for a Bayesian inference algorithm using a Virtex-5 FPGA. Although the MARC system was almost three times slower than a fully optimized FPGA solution, the design time and manual optimization effort was significantly reduced. The authors believe that the performance degradation caused by constraining the FPGA to an execution template could be overcome by application-specific customization of the architecture. In another work closely associated with MARC, Fletcher et al. 0 have implemented the Bayesian inference algorithm across several FPGAs and GPGPUs to enunciate the efficiency gap between the two platforms. Both implementations use the high-level architectural template of a GPGPU. However, application-specific logic is added to the FPGA design, requiring the user to repurpose the implemented hardware for every application. Implementation results show a ~3x performance benefit in favor of a Virtex-5 155T FPGA, as compared to the latest Nvidia Fermi-based GPGPU. In comparison, our architecture requires no application-specific logic to be embedded within the FPGA, but the user can customize the architectural parameters based on application needs. Kingyens et al. [22] have proposed a GPU-inspired soft processor programming model. The soft processor architecture exhibits several GPU design constructs including multiple processors, multithreading and vector instructions. Their work provides insight on how to best architect a GPU-inspired soft processor for maximizing the benefits of FPGA acceleration. Unlike a soft processor, our work targets FPGAs for the implementation of an actual GPGPU design based on the Nvidia G80 architecture.

The FCUDA design flow developed in [23], efficiently maps parallel CUDA kernels to customized multi-core accelerators on an FPGA. Initial performance results show that the FPGA accelerators outperform the GPU by 2x, primarily due to custom data paths and bit width optimizations. However for every new application, the CUDA program has to be re-compiled and re-synthesized onto the FPGA making it a cumbersome process. In our work, the GPU architecture needs to be synthesized on an FPGA only once, thus eliminating the need to re-synthesize hardware for different applications. Recently, Altera announced a development program on an OpenCL framework for FPGAs [24]. OpenCL is a parallel programming language based on C constructs. Altera's OpenCL program combines the OpenCL standard with the parallel performance capability of FPGAs to enable powerful system acceleration. The OpenCL compiler translates the high-level description of the user program into multicore accelerators for FPGAs, as illustrated in Figure 7. Initial benchmark results have shown that the OpenCL framework targeting FPGA exceeds the throughput of both a CPU and GPU. In addition, FPGA design using the OpenCL standard has a significant time-to-market advantage compared to traditional FPGA development using lower level hardware description languages such as Verilog or VHDL.

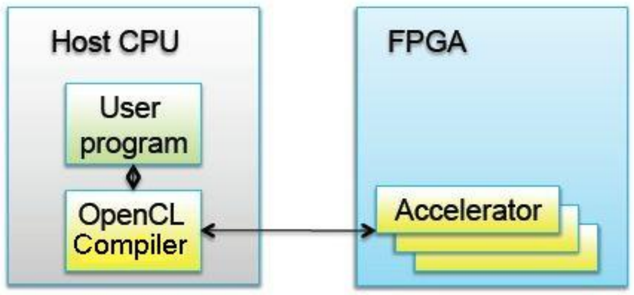


Figure 7: OpenCL framework for Altera FPGAs [24]

Although these approaches generate circuits which are optimized for a specific application and reap the associated area, performance, and energy benefits, they all require the substantial compile time associated with FPGA synthesis, mapping, and place and route. The migration of a new application to the FPGA requires substantially more time than the few seconds normally found when targeting CUDA programs to GPUs. Our goal is to reduce this time gap by effectively supporting the CUDA programming environment available to GPU programmers on FPGAs, without the costly hardware compilation typically required for reconfigurable logic. We envision such a system as being particularly useful for environments such as cloud computing or embedded systems deployed on a field, where compute nodes demand fast reconfiguration for serving different purposes at different times. In such cases, the extra cost, complexity, or power consumption of an off-the-shelf GPU in the nodes may be unwanted or unnecessary. Our approach provides a fast solution to target these environments.

2.4 Chapter Summary

This chapter introduced the Nvidia G80 hardware architecture and the CUDA programming paradigm, both of which are prerequisites for understanding the soft GPGPU architecture. It also provided an overall motivation of our project by comparing our work with the ongoing research in the field of collaborative FPGA-GPU computation for data parallel applications. In the next chapter, we shall see the architectural features of the implemented soft GPGPU.

CHAPTER 3

SOFT GPGPU ARCHITECTURE

In this chapter, we present the high-level overview of the soft GPGPU architecture. The hardware execution flow of a kernel on the soft GPGPU is enunciated, followed by detailed descriptions of the different design blocks present in the system. Towards the end of the chapter, supported CUDA instructions are described.

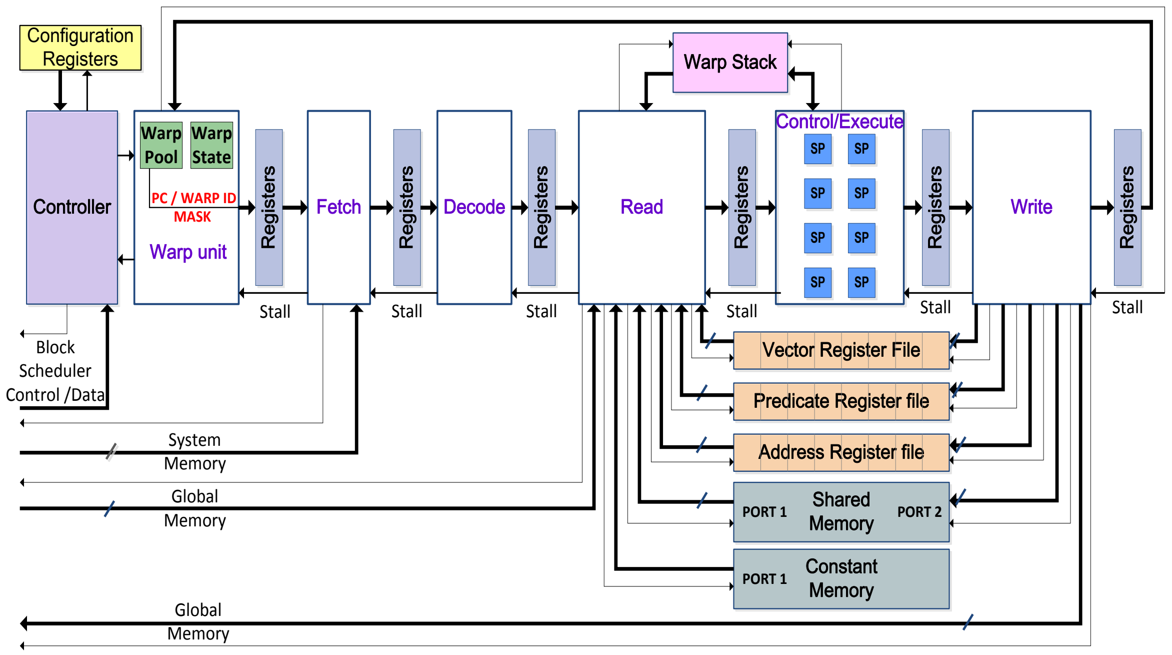


Figure 8: Streaming multiprocessor

3.1 High-level execution flow

As discussed in section 2.2.1, the Nvidia G80 architecture is made up of an array of streaming multiprocessors or SMs. Alongside the SMs, the architecture consists of a block scheduler which feeds thread blocks to the SMs, a system memory to store the kernel instructions, and a global memory to store the input and output data. As majority of the computation space is occupied by an SM, it is of particular interest to closely study its architecture. Figure 8 shows the high-level architecture of one SM. Due to the scalable nature of the soft GPGPU architecture, multiple SMs can be instantiated for more processing power by trading off the physical resources of the FPGA. The SM is designed as a five stage pipeline similar to the MIPS architecture. However, unlike MIPS, CUDA supports a *register memory architecture* allowing operations to be performed on memory as well as registers. This requires the Read stage to precede the Execute stage in order to read the operands from either memories or registers before proceeding to data execution.

GPU-based heterogeneous computing platforms consist of a host, generally a CPU, and the GPU device. During the execution of the program when the host encounters a GPU kernel call, it directs the CUDA driver API to configure the GPU for kernel execution. During configuration, the CUDA driver loads the initial kernel parameters such as the block and grid dimensions, the number of blocks per SM, the number of registers used per thread and the shared memory size. Additionally, it also populates the shared memory with user parameters—for e.g. the width of the matrices in case of matrix multiplication kernel. For independent testing and validation of the soft GPGPU system without a host, these configuration parameters are hard-coded into configuration registers prior to kernel execution. Upon encountering a kernel call, the CUDA driver is also responsible for loading the kernel into the GPU's instruction memory. We mimic this action by pre-storing the kernel on the system memory prior to execution. In the future, we envision that the host-GPU interaction will be enabled with a MicroBlaze soft processor [25] as a host, and a custom software driver to automatically populate the configuration registers and the instruction memory with the CUDA kernel.

After the configuration process is finished, the block scheduler schedules thread blocks to the SM with each block identified by its block ID. The block scheduler then passes the relevant control and data information of the scheduled blocks to the controller. The controller acts as the interface between the block scheduler and the SM. As per CUDA requirements, the controller performs two operations:

1. It populates the first 16 bytes of the shared memory using block scheduler information.
2. It writes all the register *R0*s in the vector register file corresponding to different threads with their respective thread IDs.

Following this, the warp generation and warp scheduling processes are initiated as detailed in the next section.

3.2 Pipeline description

This section describes the various pipeline stages in the order that an instruction would flow through the pipeline.

3.2.1 Warp unit

As mentioned in section 2.2.2, CUDA threads in an SM are launched in groups known as *warps*. The warp unit is responsible for generating these warps and scheduling them in a round-robin fashion. Each warp contains data and an associated state. The warp data primarily holds the warp ID ranging from value 0 to (*maximum warps* -1), the program counter (PC), and a thread mask. The thread mask is particularly useful during conditional execution to mask out threads within a warp that do not lie on the current execution path. Each warp maintains its own PC and thus is independent to take its own path. The mask size is same as the warp size, i.e. 32 bits. The warp state indicates the status of the warp which can either be Ready, Active, Waiting or Finished. The Ready state indicates that the warp is idle and is ready to be scheduled. Active state indicates that the warp is currently active in the pipeline. In order to synchronize warps within a block, CUDA supports explicit barrier synchronization instructions. Warps that reach the barrier instruction first have to wait for other warps to reach to the same checkpoint, and hence are marked as Waiting. When all the threads in a warp finish executing the kernel, the warp is declared as Finished. Within a warp, threads are arranged in rows depending on the number of scalar processor (SP) instantiated within an SM. For e.g. for an 8 SP configuration, a warp would be arranged in four rows with each row containing 8 threads. Similarly, for a 16 SP configuration, a warp would be arranged in two rows with 16 threads each. The maximum parallelism is achieved with 32 SPs and one row.

In our architecture, the warp data and state are stored on the FPGA taking advantage of dual-ported Block RAMs. The warp data is stored in a warp pool memory and the warp state is stored in the warp state memory. Both memories are indexed using the warp ID. Initially, all the warp data are generated using the respective warp IDs, PC pointing to first kernel instruction address, i.e. 0x00000000, and an instruction mask with all threads active, i.e. 0Xffffffff. The warp state is initialized to Ready for all the warps. Once the warp generation is complete, the data for the first warp is read from the warp pool, its state is verified as Ready and all its rows are scheduled one after another. Likewise, other warps are scheduled one after another every cycle. This scheduling process is handled by the warp scheduler. A scheduled warp is primarily recognized by its warp ID, PC and thread mask. The PC, thread mask and the warp state are updated in the corresponding memories every time a warp reaches back to the warp unit stage after looping through the entire pipeline.

This normal flow of warp scheduling is somewhat interrupted in case of the barrier synchronization instruction. A warp executing this instruction through the pipeline is marked Waiting towards the end of the pipeline. A fence register is maintained to register incoming warps that are in the Waiting state. The synchronization flow is as shown in Figure 9. The width of the fence register is equal to the total number of warps per block. For every incoming warp in the Waiting state, the warp unit sets the fence register bit corresponding to the warp. It then reads the fence register to check if all the bits are set which would indicate the arrival of all warps that are in the Waiting state. If the condition is true, all warps are synchronized and the barrier is released. The warp unit changes all the warp states to Ready and normal warp scheduling resumes.

3.2.2 Fetch and decode stage

The fetch stage fetches the binary instructions based on the warp PC forwarded by the warp unit. The fetched instruction can be visualized as being mapped onto all the threads in the row, SIMD style. The CUDA instruction can be either 4 bytes or 8 bytes depending on whether it is a short or a long instruction respectively. After fetching the instruction, the PC value is incremented (by 4/8 bytes) to point to the next instruction. The decode stage decodes the binary instruction to generate several output tokens such as the instruction type, instruction length, source and destination operands, data types, conditional execution, etc.

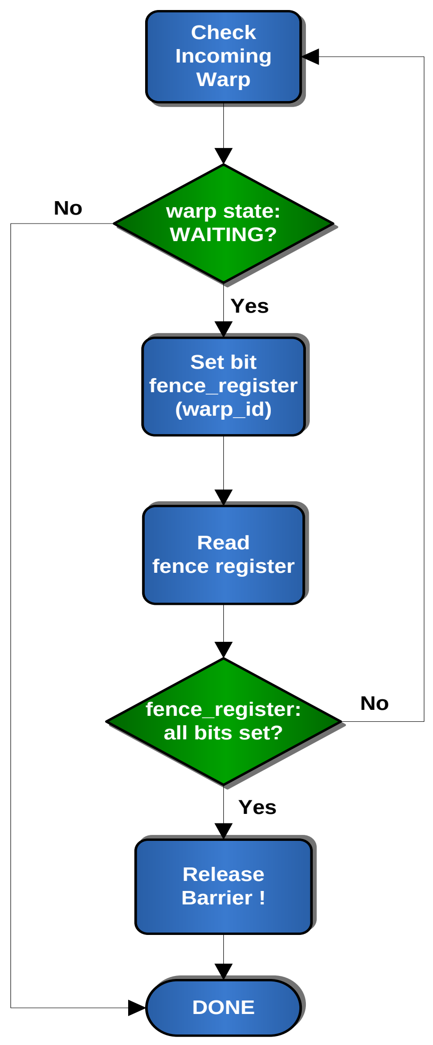


Figure 9: Barrier synchronization using fence registers

3.2.3 Read stage

In the read stage, source operands are read from register files and memories depending on the decoded inputs. The vector register files are implemented as register file banks such that each thread has its own set of registers. The vector register file is used to store general-purpose registers. Threads in a warp are mapped to the vector register file as shown in Figure 10. Each thread within a row is mapped to a different register file for reading and writing data in parallel. To differentiate between threads lying in the same column but in different rows, each register file is split into 4 memory banks. Each bank is implemented as a dual port memory and the decoded row ID is used to choose a particular memory bank. The size of a memory bank is determined by the total number of warps and the total number of registers used by each thread. For the benchmarks under consideration, it was found out that the maximum number of registers used by any application was 12. For accommodating registers for all 24 warps, a memory bank must be able to hold 24 x 12 = 288 registers. If each register is 4 bytes long, we need a memory bank size of 288 x 4 = 1152 Bytes. The register file was physically implemented on the FPGA using the on-chip BRAMs of size 1152 Bytes.

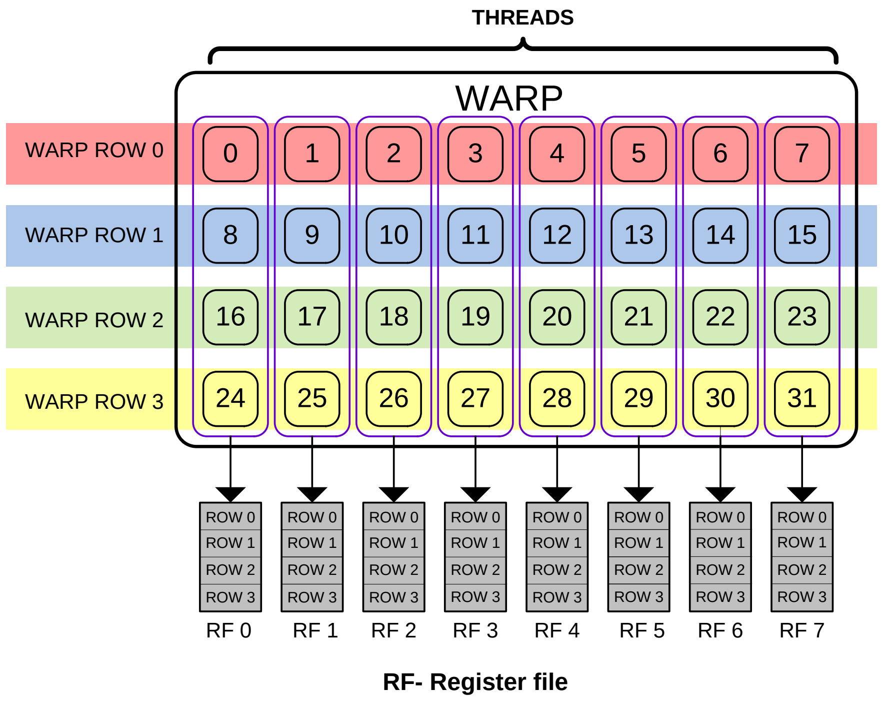


Figure 10: Vector register file read operation

The address registers and predicate registers are also mapped in the same fashion as the vector registers. The address register file stores the memory offsets for gather-scatter memory operations. Gather-scatter operations are same as load-store operations, but in burst mode i.e. data is read in bursts rather than sequentially. Each thread is allotted four address registers. The predicate register file holds *predicate* flags used for branches and conditionally executing instructions (predication). Predicate flags store different branching conditions like zero, non-zero, sign, overflow, carry, etc. Instructions prefixed with a predicate flag are termed predicated instructions [26].

The shared, constant and global memories are implemented using dual port BRAMs [27] with one port for the Read stage and the other for the Write stage. This ensures that the Read and Write stages can access the memories simultaneously in the pipeline. The total shared memory space is divided between different blocks per SM and has a total size of 16 KB. The constant memory is 8 KB read-only memory used to store constant data. The global memory stores the input/output data and has a total size of 256 KB. Unlike the standard MIPS architecture where the memory address is calculated in the Execute stage, memory controllers with dedicated address calculation units are embedded within the Read and Write stages to access data. The warp stack is used to store warp information while executing control-flow instructions. Its uses are detailed in the next section.

3.2.4 Control / Execute stage

This stage forms the crux of the soft GPGPU pipeline. It performs all the data processing (arithmetic and logical) with the help of functional units or scalar processors (SP). Each thread in the warp row is mapped to one SP enabling parallel execution. In our architecture, the number of SPs can be varied for more or less processing power. The available configurations of the SPs are 8, 16 and 32. Currently, the SPs support only integer type operations like addition, subtraction, multiplication, multiply and add, data type convert, bit shifting and logical operations such as AND, OR, NOR, XOR, etc. All these operations are implemented using Matlab Simulink models [28] which are converted to HDL code using the Xilinx System Generator [29].

The control unit is responsible for executing all control flow instructions which include conditional and unconditional branches, barrier synchronization, kernel return, and set synchronization point. In the case of branch instructions, the control unit pushes the current warp data onto the stack and executes one of the branch paths. Upon finishing execution of the path, the warp data is popped off the stack for executing the other branch path. In case of a barrier instruction, the control unit marks the warp state as waiting. The synchronization is then taken care of by the warp unit as explained earlier in the chapter. The return instruction signifies the end of kernel. If all the threads in a warp execute this instruction (no threads are masked), the warp is killed, i.e. marked as Finished. Finished warps are no longer scheduled by the warp unit. The set synchronization instruction is used before potentially divergent branches. A warp is said to diverge if the branch outcome is not same for all threads in the warp. The set synchronization instruction is used to set the reconvergence point of a branch – an instruction that will be reached irrespective of whether or not the branch is taken. The synchronization point is set by pushing the reconvergence PC onto the stack. In case of divergence, execution proceeds along one path (say, taken) until the reconvergence point is reached. When the point is reached, the execution switches back to the other path (not-taken). When the reconvergence point is reached for a second time, the reconvergence PC is popped off the stack and normal thread execution continues from the reconvergence instruction and beyond. Figure 11 explains the sequence of operations that are performed to handle branch divergence.

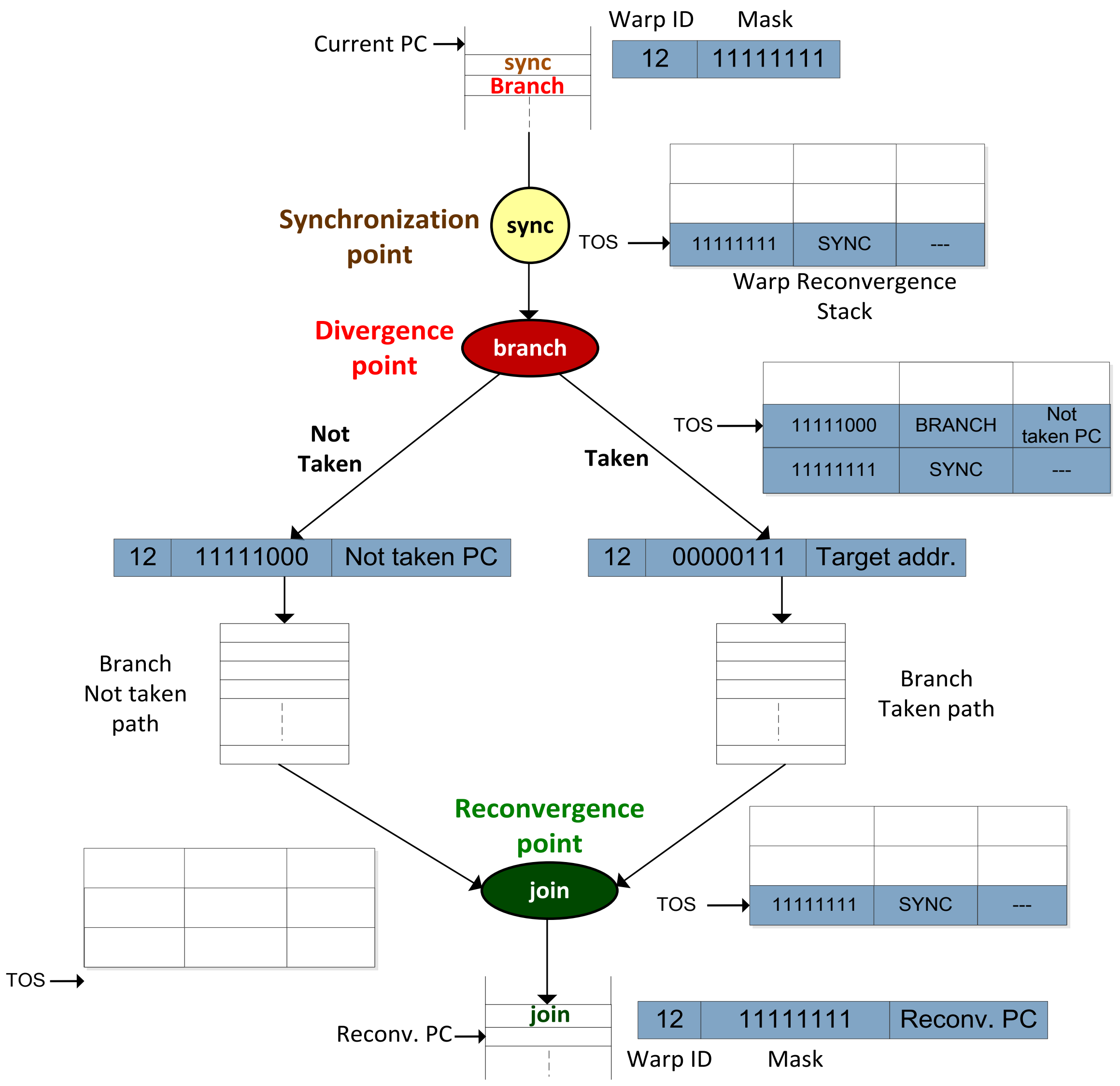


Figure 11 : Handling branch divergence

For the sake of simplicity, consider the case in which we have (say) only eight threads in a warp. Figure 11 shows the scenario when execution is just about to hit a diverging branch. As discussed before, the synchronization instruction precedes the diverging branch to set the synchronization point. The join instruction at the end is the reconvergence point. The thread mask is equal to the warp size, i.e. 8 bits wide. The warp convergence stack is a hardware structure that keeps track of diverged branches. There is one stack per warp. Each entry in the stack has three fields—the thread mask, control flow opcode and the next PC. Assume all the threads are active before diverging, i.e. the initial thread mask = “11111111”.

i. Execution reaches the synchronization point. The stack is populated with the current thread mask and the control opcode SYNC.

ii. Execution reaches the divergent branch. Either of the branch paths can be taken first. If (say) branch *taken* is being executed first, the thread mask of the *not taken* path (compliment of the *taken* mask), the control flow opcode i.e. Branch, and the *not taken* PC is populated at the top of the stack.

iii. The target address and the thread mask of the *taken* branch path are loaded by the warp scheduler in the next cycle and following instructions are executed.

iv. Execution reaches the reconvergence point for the first time. The join instruction is detected and the top-of-stack (TOS) entry is popped. The TOS pointer is decremented by one.

v. The popped thread mask and PC corresponding to the *non taken* branch are loaded by the warp scheduler in the next cycle, and following instructions are executed.

vi. Execution reaches the reconvergence point for the second time. The join instruction is detected and the top-of-stack (TOS) entry is popped. The TOS pointer is decremented by one and the stack *empty* signal goes high.

vii. The popped opcode is detected as SYNC. Consequently, the popped thread mask is loaded by the warp scheduler in the next cycle. However, instead of the loading the PC from the popped PC, the PC of the instruction next to the join instruction, called the reconvergence PC is loaded.

viii. Both the branch paths are now executed for different sets of threads, and beyond this point all threads resume parallel execution. The same control flow would also support nested branches with sync instruction before every diverging branch and join instruction at every reconvergence point.

3.2.5 Write stage

The Write stage writes the vector register file with temporary data, address register file with memory offsets, predicate register file with predicate flags, shared memory with either temporary data or results, and the global memory with final results. The sequence of operations for writing into memory and registers is exactly opposite to the Read stage. The warp data and state is looped back to the warp unit for updating the warp pool and state memories.

All pipeline stages output a *stall* signal that is fed to the preceding stage. The stall signal indicates that the stage is busy and not ready to accept new data. Every stage has to make sure that the input stall signal is low before passing its own data to the next stage.

This ensures smooth data flow from one stage to another through the pipeline and avoids data corruption across stages.

3.3 Supported CUDA instructions

The soft GPGPU supports a subset of the Nvidia G80 instruction set with compute capability 1.0 [30] . Instructions were tested based on the requirements of the selected benchmarks. A total of 27 instructions out of the 40 distinct integer instructions (that we are aware of) were tested as a part of this thesis. The list of all instructions (supported and unsupported) is shown in Table 2.

Opcode	**Description**	**Tested**
I2I	Copy integer value to integer with conversion	☑
IMUL/IMUL32/ IMUL32I	Integer multiply	☑
SHL	Shift left	☑
IADD	Integer addition between two registers	☑
GLD	Load from global memory	☑
R2A	Move register to address register	☑
R2G	Store to shared memory	☑
BAR	CTA-wide barrier synchronization	☑
SHR	Shift right	☑
BRA	Conditional branch	☑
ISET	Integer conditional set	☑
MOV /MOV32	Move register to register	☑
RET	Conditional return form kernel	☑
MOV R, S[]	Load from shared memory	☑
IADD, S[],R	Integer addition between shared memory and register	☑

GST	Store to global memory	✓
AND C[], R	Logical AND	✓
IMAD/IMAD32	Integer multiply-add; all register operands	✓
SSY	Set synchronization point; used before potentially divergent instructions	✓
IADDI	Integer addition with an immediate operand	✓
NOP	No operation	✓
@P	Predicated execution	✓
MVI	Move immediate to destination	✓
XOR	Logical XOR	✓
IMADI/ MAD32I	Integer multiply-add with an immediate operand	✓
LLD	Load from local memory	✓
LST	Store to local memory	✓
A2R	Move address register to data register	-
ADA	Add immediate to address register	-
BRK	Conditional break from loop	-
BRX	Fetch and address from constant memory and branch to it	-
C2R	Conditional doe to data register	-
CAL	Unconditional subroutine call	-
COS	Cosine	-
ISAD/ISAD32	Sum of absolute difference	-
R2C	Move data register to conditional code	-
MVC	Move form constant memory to destination	-
RRO	Range reduction operator	-
VOTE	Warp-vote primitive	-
TEX/TEX32	Texture fetch	-

Table 2: Instruction set

3.4 Chapter Summary

In this chapter, we discussed the hardware architecture and the overall pipeline execution flow of the soft GPGPU. The functionality of the different pipeline stages and other supporting modules were described in context of the CUDA programming model. The supported instruction set was also presented. In the next chapter, we shall examine testing aspects of some of these blocks and validation of the soft GPGPU system.

CHAPTER 4

TESTING AND VALIDATION

This chapter presents the testing and validation aspects of the prototype soft GPGPU design. The first section focuses on the testing methodology for the architecture. We describe our testing approach and discuss results that are of particular interest. To conclude, we present the validation flow, the involved methodology and validation results.

4.1 Software flow

The software flow for executing a CUDA kernel on the soft GPGPU is as shown in Figure 12. The left portion of the figure illustrates the software flow as apparent to the user. The process is split up into two phases as compile-time and run-time. During compile-time, the kernel is fed to the Nvidia CUDA compiler (*nvcc*) which converts it to parallel thread execution (PTX) code. PTX is a low-level assembly–like programming language that exposes the GPU as a data-parallel computing device [26]. It defines a stable programming model and a *virtual* instruction set architecture (ISA) for Nvidia GPUs. The PTX does not directly represent the machine instruction set, but is only an intermediate language that is compiled to target-specific assembly instructions. During run-time, the PTX assembly is passed to the CUDA driver API (Application Programming Interface). The driver API then converts the PTX to a CUDA binary (*.cubin*) which is targeted to the soft GPGPU. As we are not targeting actual Nvidia hardware, we use the runtime libraries provided by Nvidia to mimic the driver functionality.

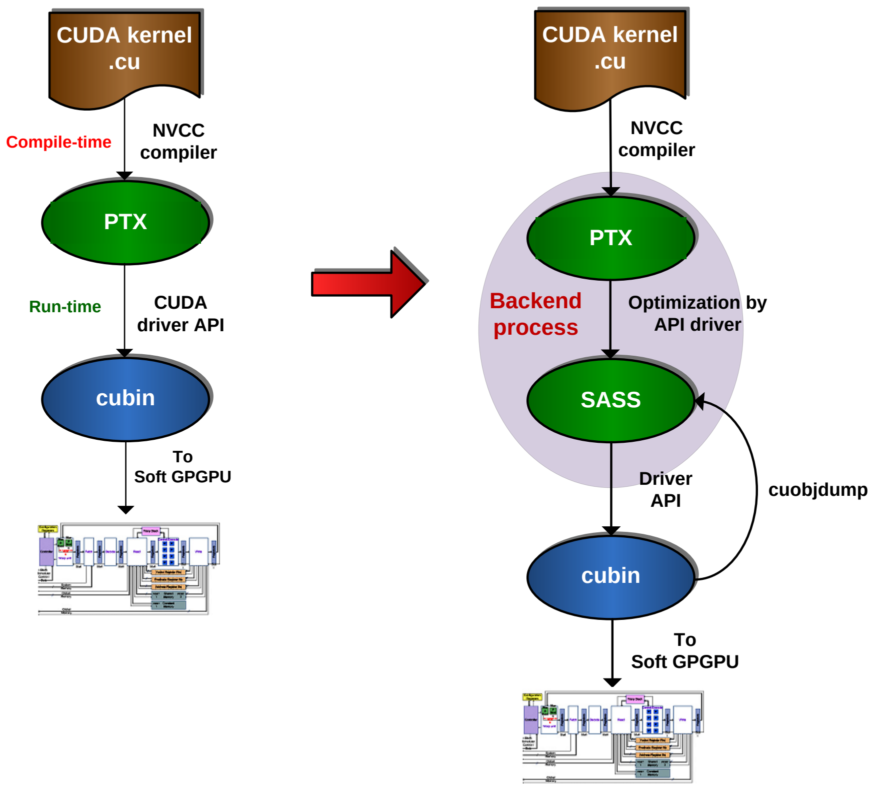


Figure 12: (Left) Software flow as apparent to the user, (Right) Actual software flow which generates SASS

In order to test and validate the soft GPGPU, it is necessary use the hardware assembly instructions that correspond to the generated binary. As noted earlier, the PTX assembly is only an intermediate language and does not map to actual hardware instructions executed on the GPU. Thus, the PTX cannot be used as the golden reference. Further investigation into the CUDA compilation flow revealed that during runtime, the driver API converts the PTX instructions to another format called Source and Assembly (SASS) [31], as shown on the right in Figure 12. SASS is specific to the target GPU architecture and represents native assembly instructions that are executed on the Nvidia hardware. However, it is interesting to note that the PTX-to-SASS conversion is not directly visible to the user and stays as a backend process. In order to generate the SASS instructions, the CUDA binary is disassembled using the *cuobjdump* [32] utility provided by Nvidia which can then be used for testing and validation purposes.

Microsoft Visual Studio 2008 and Nvidia Toolkit v2.3 [33] are integrated together for this compilation process. The Nvidia toolkit is comprised of the Nvidia CUDA compiler (*nvcc*), and the CUDA driver and runtime API libraries. It supports integration with Visual Studio 2008 by providing Nvidia compilation rules for building CUDA applications.

4.2 Testing experiments

A simulation-based approach is adopted for testing the different design blocks. Testbenches are generated using either hand-modeled test cases or by using the binary instructions (for the decode stage). The design is then subjected to logic simulation using these testbenches. A typical verification flow using logic simulation is as shown in Figure 13. The requirements drive the development of the RTL model and it influences the verification plan for developing the testbench. The verification plan consists of the test cases to be taken into consideration while generating the testbench. The simulation tool reads the testbench and the RTL model for running the simulation process. The result of the simulation is compared with the expected outputs to infer if a bug is present in the design. In the event the result is positive, the RTL design is debugged and appropriate design modifications are made. If no bug has been found, the simulation results are examined to verify that all paths are exercised, in which case the verification process is complete.

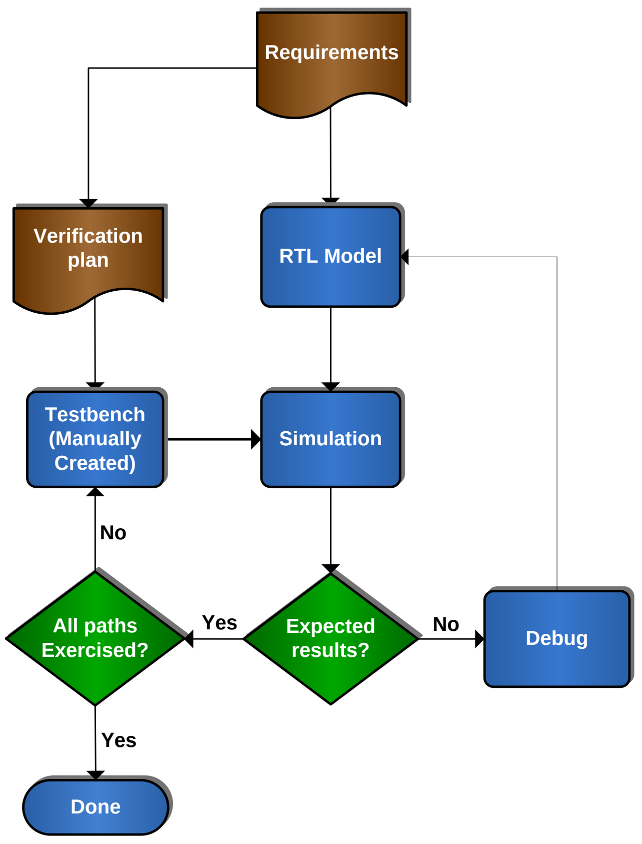


Figure 13: Simulation process for logic verification

The following sections describe the conducted testing experiments for some of the critical blocks in the system. The simulations were carried out using the ModelSim SE 10.0 simulator [34].

4.2.1 Decode stage

The decode stage was one of the more challenging blocks to design and test in the system. Nvidia does not reveal the G80 microarchitecture for proprietary reasons, as a result of which there is limited amount of available information on the binary mapping of assembly instructions. In order to closely understand the assembly instructions of the G80 architecture, *decuda* [35], a CUDA binary disassembler was used as a reference. Additional cues were taken from academic GPGPU simulators like Barra [36] and

GPGPU-Sim [31] [37] to design the decode stage. The primary design of the decode stage is as shown in Figure 14.

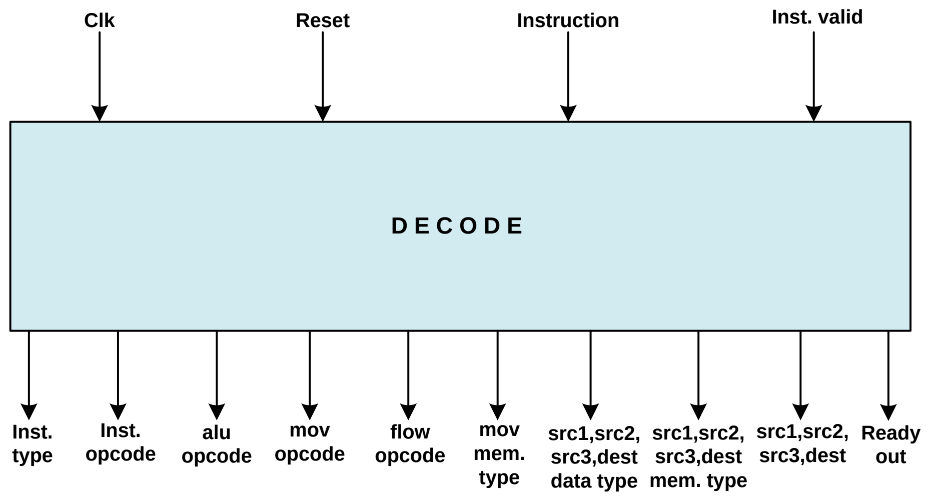


Figure 14: Decode stage

i. *Inst. type* represents the instruction length (full – 64 bits, half – 32 bits).

ii. *Inst. opcode* signifies the instruction type.

iii. *alu opcode, mov opcode* and *flow opcode* represent the subtypes for each opcode type.

iv. *mov mem. type* represents the type of data transfer. It can either be between two registers or between a register and a memory.

v. *src1, src2, src3, dest data type* represent the source and destination data types.

vi. *src1, src2, src3, dest mem. type* represent the source and destination memory types.

vii. *src1, src2, src3, dest* are the source and destination numbers

For experimentation, a preliminary CUDA kernel was written, and the corresponding binary and SASS instructions were generated as shown in Figure 16. The __*global*__ keyword specifies that the function is executed on the GPU. The kernel reads a one-dimensional integer array '*a'* containing *N* elements and multiplies each element by a factor of 2. *blockIdx.x* represents the block ID, *blockDim.x* represents block dimension in terms of number of threads and the thread ID is represented by *threadIdx.x*. These parameters are used to calculate the distinct indices of the array that each thread would access individually.

The decode results are illustrated in Figure 15. By manually comparing the decode outputs against the SASS assembly reference shown in Figure 16, correct decode operation was verified. Several instructions from other academic resources [38] were used to exercise the decode stage and necessary design modifications were made. Some of the bugs were also discovered and rectified while simulating actual CUDA benchmarks described in the later part of this thesis

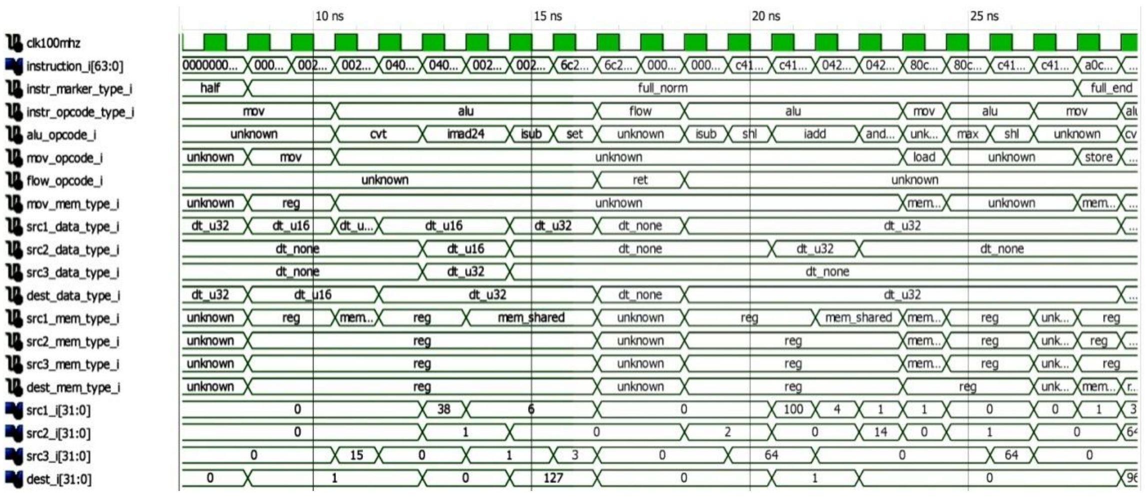
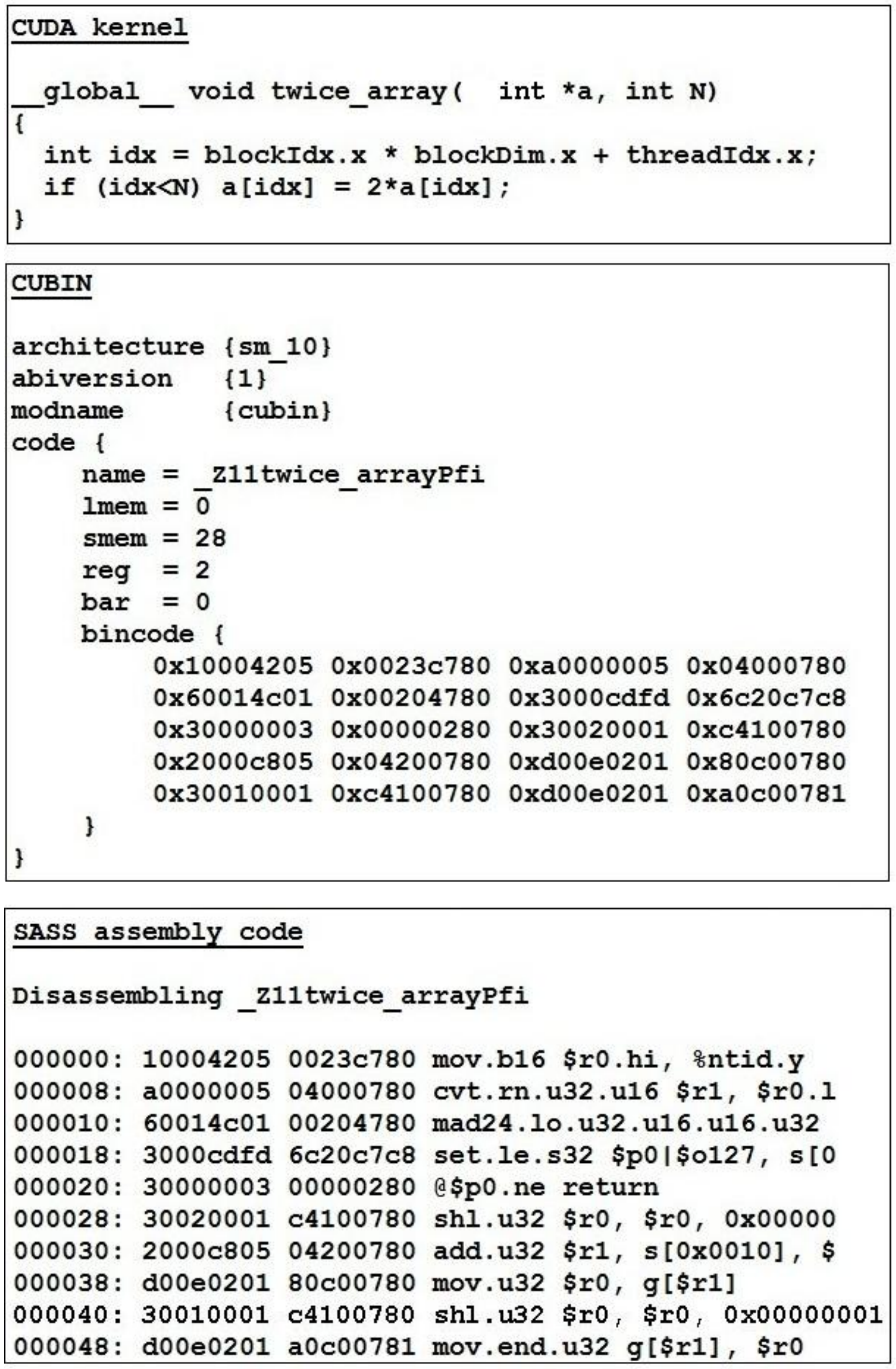


Figure 15: Decode stage results



```
CUDA kernel

__global__ void twice_array(  int *a, int N)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx<N) a[idx] = 2*a[idx];
}
```

```
CUBIN

architecture {sm_10}
abiversion   {1}
modname      {cubin}
code {
    name = _Z11twice_arrayPfi
    lmem = 0
    smem = 28
    reg  = 2
    bar  = 0
    bincode {
        0x10004205 0x0023c780 0xa0000005 0x04000780
        0x60014c01 0x00204780 0x3000cdfd 0x6c20c7c8
        0x30000003 0x00000280 0x30020001 0xc4100780
        0x2000c805 0x04200780 0xd00e0201 0x80c00780
        0x30010001 0xc4100780 0xd00e0201 0xa0c00781
    }
}
```

```
SASS assembly code

Disassembling _Z11twice_arrayPfi

000000: 10004205 0023c780 mov.b16 $r0.hi, %ntid.y
000008: a0000005 04000780 cvt.rn.u32.u16 $r1, $r0.l
000010: 60014c01 00204780 mad24.lo.u32.u16.u16.u32
000018: 3000cdfd 6c20c7c8 set.le.s32 $p0|$o127, s[0
000020: 30000003 00000280 @$p0.ne return
000028: 30020001 c4100780 shl.u32 $r0, $r0, 0x00000
000030: 2000c805 04200780 add.u32 $r1, s[0x0010], $
000038: d00e0201 80c00780 mov.u32 $r0, g[$r1]
000040: 30010001 c4100780 shl.u32 $r0, $r0, 0x00000001
000048: d00e0201 a0c00781 mov.end.u32 g[$r1], $r0
```

Figure 16: Sample CUDA kernel and corresponding cubin, SASS code

4.2.2 Read / Write stage

The Read and Write stages were verified together by interfacing them in a tandem fashion with the register files and the memories as shown in Figure 17. The written and read data are compared against each other to verify correct read-write operation. The design of both the stages includes finite state machines (FSM), where each state represents a register or a memory operation. The verification testbench is designed such that all states of the FSM are traversed at least once in both the stages. It was noted that

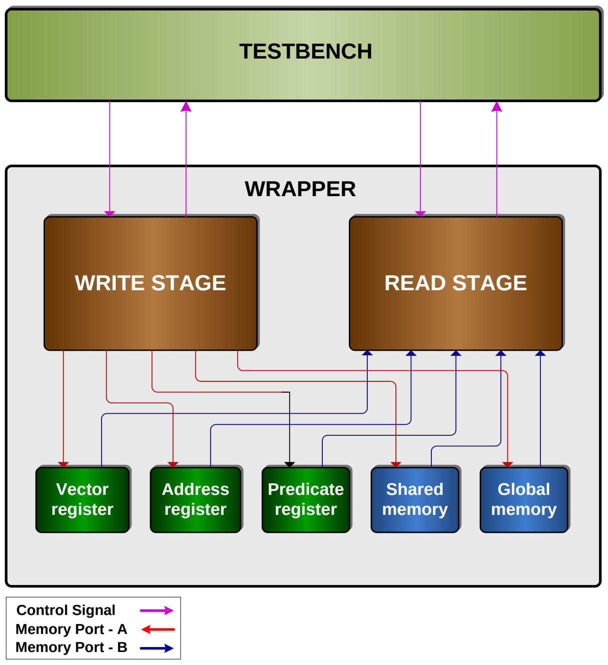


Figure 17: Read-Write verification structure

the sequence of operations for writing and reading the global/shared memory exercises the register files as well, as shown in Figure 18. The sequence is initiated by writing the address registers that hold the memory offsets for each thread. This is followed by writing the vector registers to store the base address of the memory. In the next step, the base address and the offset are read and combined together to calculate the effective memory address – the address used for writing the memory. Following the

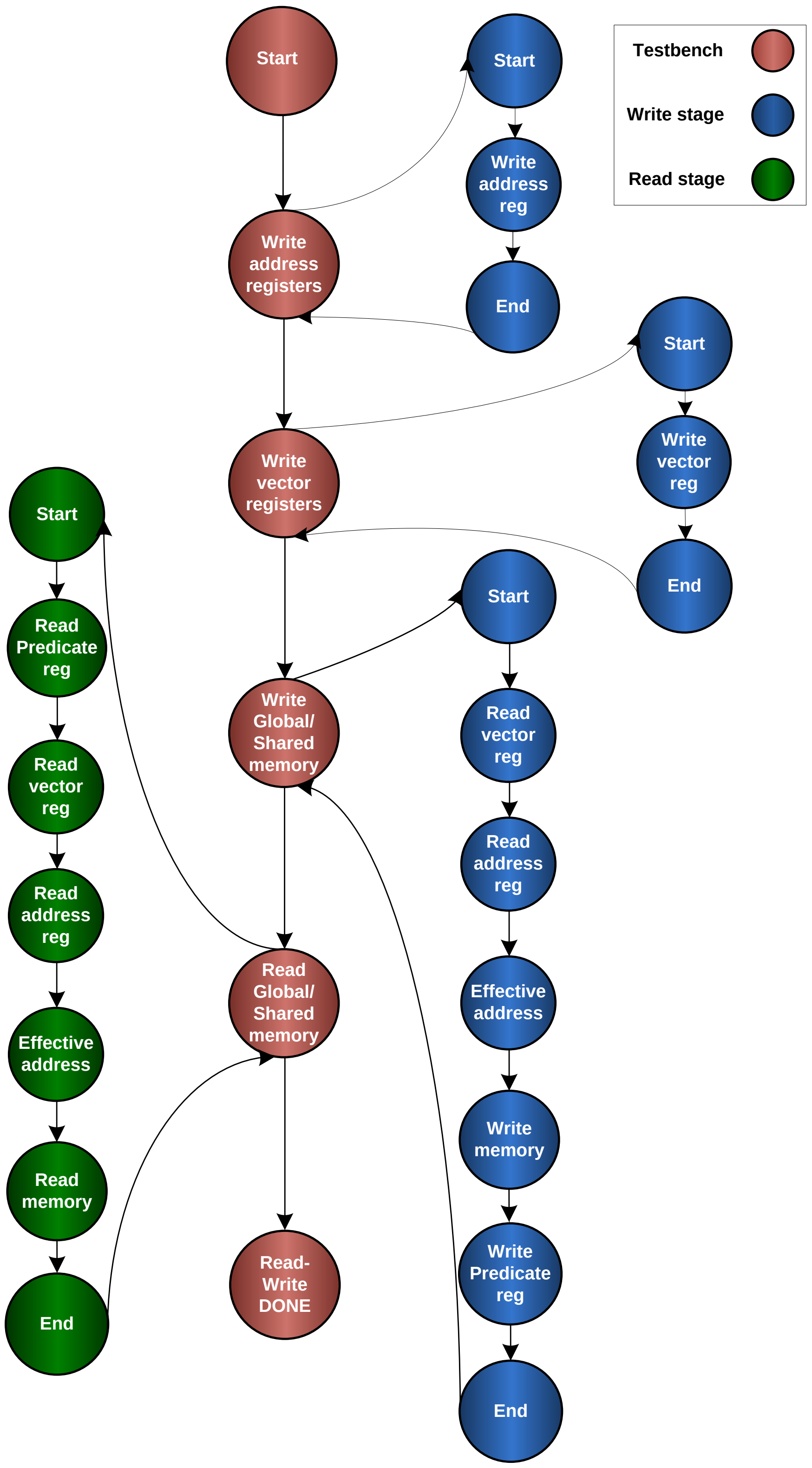


Figure 18: Read-Write verification FSM

memory write operation, predicate flags (though not necessary for a memory operation) are written into the predicate register file. The read stage FSM is initiated by reading the predicate register file. This state is traversed at the beginning of every read cycle during the execution of predicated instructions. Following the predicate register read, the same sequence of operations are repeated to calculate the effective memory address read back the data from memory. The written and read value of the registers and memory are compared at different stages of the FSM to verify accurate read-write operations.

Figure 19 shows the simulation result for the global memory write stage. The *global_memory_cntrl_state_machine* signal represents the state of the global memory controller. As illustrated by this signal, the effective address is calculated in the beginning using the vector and address registers, followed by scatter write operation to the global memory. The *gmem_addr_i* and *gmem_wr_data_i* (last two signals) represent the effective address and the data written to the memory, respectively.

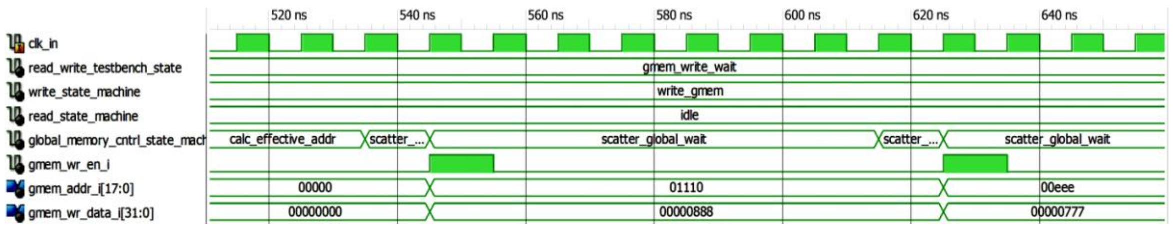


Figure 19: Global memory write

The read stage simulation result for the global memory is shown in Figure 20 and follows similar sequence of operations as the write stage. The *gmem_addr_i* and *gmem_rd_rd_data_o* (last two signals) represent the address and the data read back from the memory, respectively.

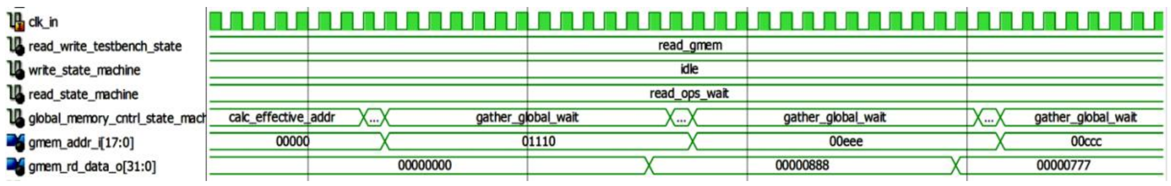


Figure 20: Global memory read

Comparing the two results, the read-write operations for global memory, vector register and the address register file are verified. Other results are omitted for the sake of brevity as the shared memory operations are exactly the same as global memory, whereas the inherent effective address calculation testifies correct register file read-write operations.

4.3 System validation

The soft GPGPU design was validated by benchmarking the platform with five standard CUDA applications that are described in the next section. The basic validation flow is as shown in Figure 21. The CUDA kernels were compiled using the NVCC compiler and the original binaries were executed on the soft GPGPU without any code modifications. Counterpart C/C++ applications were compiled using standard GCC compiler and executed on an x86 platform. The results generated from the C/C++ execution were considered as the golden reference for comparison. The ModelSim simulation results generated for all the benchmarks were found to be accurate, thus validating correct soft GPGPU functionality.

4.3.1 Benchmark suite

Exhaustive validation experiments were conducted across a suite of five CUDA benchmarks as shown in Table 3. The benchmarks were procured from several academic resources. The MatrixMul and Transpose benchmarks were taken from the CUDA

Programming Guide [4]. Bitonic sort was procured from Duke University [39]. Autocor benchmarks were procured from the University of Wisconsin-Madison [40]. The Reduction benchmark was obtained from the University of Notre Dame [41].

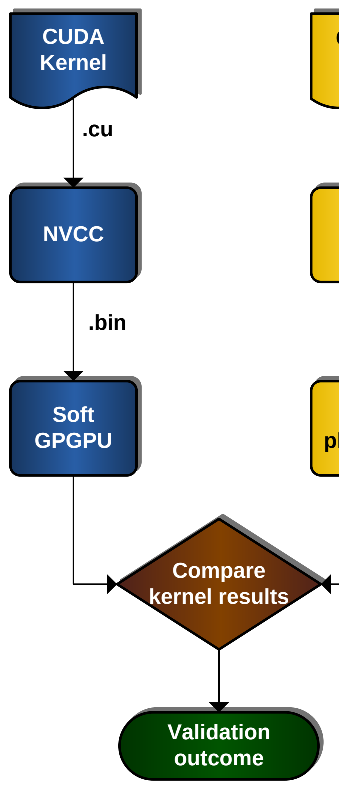


Figure 21: Validation flow

All benchmarks are restricted to integer data type. The selection criterion was based upon their popularity in the GPGPU research community. The assortment of highly data-parallel and control-flow intensive benchmarks, help us fairly evaluate our platform for applications with different characteristics. Bitonic is the most control-flow intensive, while Autocor has some control flow. Reduction, MatrixMul and Transpose are fairly data parallel.

Benchmark	**Description**	**Sizes of tested datasets**	**Percent of supported ISA used**
Autocor	Autocorrelation of 1D array	16,32,64,128,256	69.2%
Bitonic	High performance sorting network	16,32,64,128,256	57.7%
MatrixMul	Multiplication of square matrices	16x16,32x32,64x64, 128x128,256x256	69.2%
Reduction	Parallel reduction of 1D array	16,32,64,128, 256,512	61.5%
Transpose	Matrix transpose	16x16,32x32,64x64, 128x128,256x256	53.8%

Table 3: Benchmark suite

- **Autocorrelation:** Autocorrelation is the correlation of a signal with itself. The basic equation for autocorrelation of a discrete-time signal is shown below:

$$r_{xx}(l) = \sum_{n=-\infty}^{n=\infty} x(n)x(n-l) \qquad\qquad l = 0, \pm 1, \pm 2, \ldots$$

  It basically consists of a series of Multiply and Add operations. The Autocor operation can be parallelized by having each thread compute an element of the autocorrelation array

- **Bitonic sort:** Bitonic sort is one of the fastest sorting networks. A sorting network consists of sequence of comparisons that is data-independent. This makes sorting networks suitable for hardware implementation on parallel processing platforms.

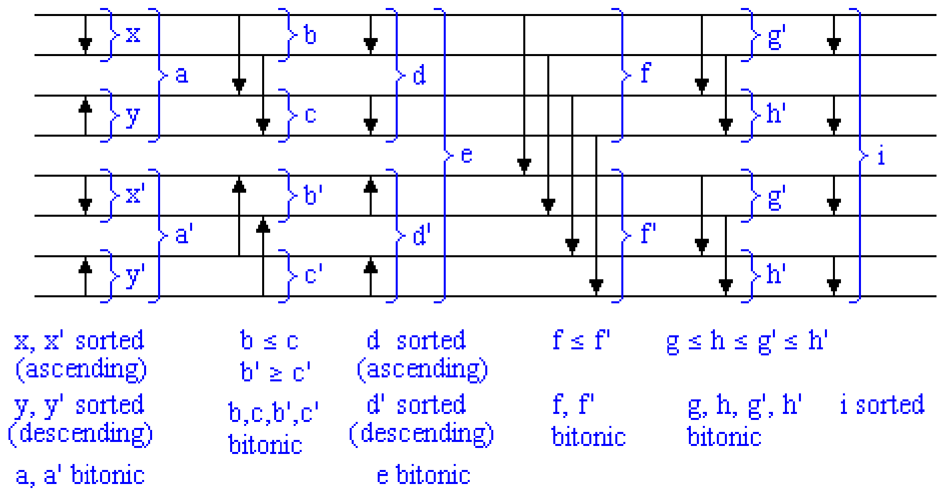


Figure 22: Bitonic sorting network [42]

The ascending bitonic sort network is shown in Figure 22. For an array of size *n*, the bitonic network consists of $\Theta(n \cdot log(n)^2)$ comparison operations through $\Theta(log(n))$ stages, with each stage performing *n/2* comparisons. The head of the arrow points to the larger of the two values. Passing through the network, all the values at the input are sorted in an ascending order at the output, as they pass through the network. Considering the structure of the network, the comparison operations in each stage can be parallelized, ideally leading to an *n/2* speedup.

- **Matrix multiplication:** This benchmark multiplies two square matrices with integer data type. The application can be parallelized by computing each element of the product matrix in parallel.
- **Reduction:** A reduction algorithm basically extracts a value from an array by performing an array operation. The operation can be sum, min, max, average etc. In our case, we have chosen the summation operator which sums all the elements of the array. A basic reduction network is shown in Figure 23. Though the

reduction network looks simple, there are a lot of opportunities to parallelize the CUDA kernel in a way that exploits maximum benefits.

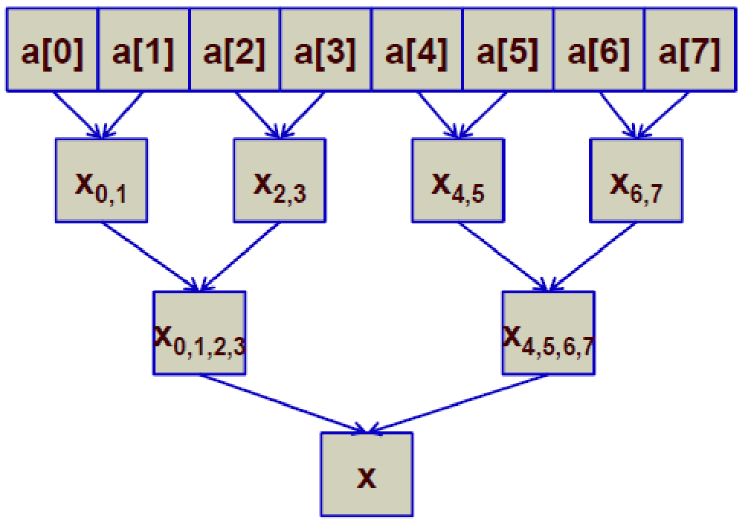
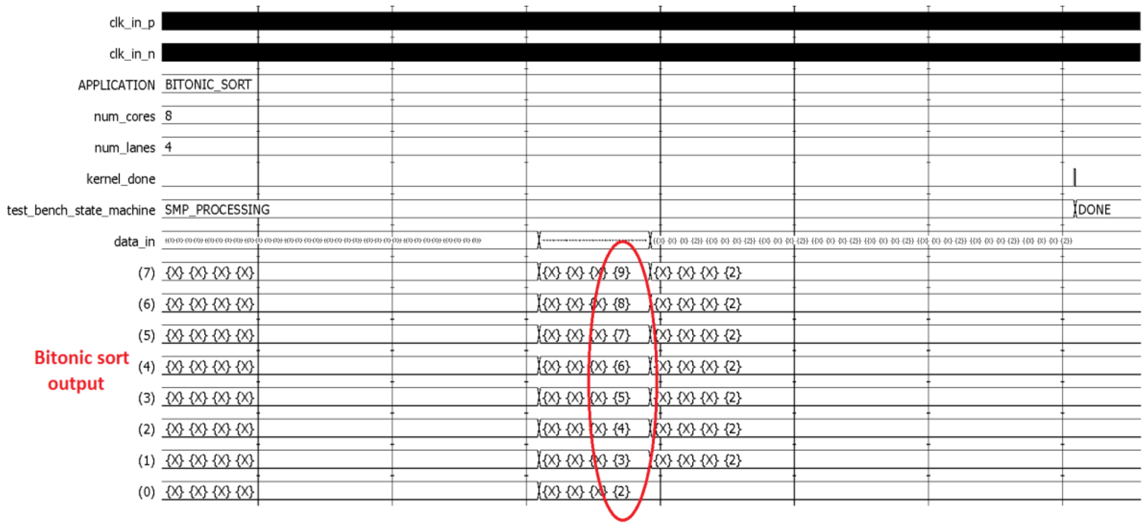


Figure 23: Parallel reduction network [43]

- **Transpose**: This benchmark computes the transpose of an integer square matrix. It is parallelized such that each matrix element is computed in parallel.

4.3.2 Validation results

Benchmarks were simulated for dataset sizes shown in Table 3. As an example, the soft GPGPU simulation result for the Bitonic benchmark is shown in Figure 24. The results show a sorted array {9, 8, 7, 6, 5, 4, 3, 2} of eight integers as indicated by the red ellipse. As another example, the simulation result of the Reduction benchmark is shown in Figure 25. The size of the array was fixed to 512 elements with the array values {0, 1, 2, 3, 4, 5, 6, 7, 0, 1, 2, 3, 4, ….and so on}. For such an array, the expected sum is 1792, as shown by the red circle in the figure.



clk_in_p
clk_in_n
APPLICATION BITONIC_SORT
num_cores 8
num_lanes 4
kernel_done
test_bench_state_machine SMP_PROCESSING
DONE
data_in
(7) {X} {X} {X} {X} {X} {X} {X} {9} {X} {X} {X} {2}
(6) {X} {X} {X} {X} {X} {X} {X} {8} {X} {X} {X} {2}
(5) {X} {X} {X} {X} {X} {X} {X} {7} {X} {X} {X} {2}
(4) {X} {X} {X} {X} {X} {X} {X} {6} {X} {X} {X} {2}
(3) {X} {X} {X} {X} {X} {X} {X} {5} {X} {X} {X} {2}
(2) {X} {X} {X} {X} {X} {X} {X} {4} {X} {X} {X} {2}
(1) {X} {X} {X} {X} {X} {X} {X} {3} {X} {X} {X} {2}
(0) {X} {X} {X} {X} {X} {X} {X} {2}
Bitonic sort output

Figure 24: Bitonic sort result

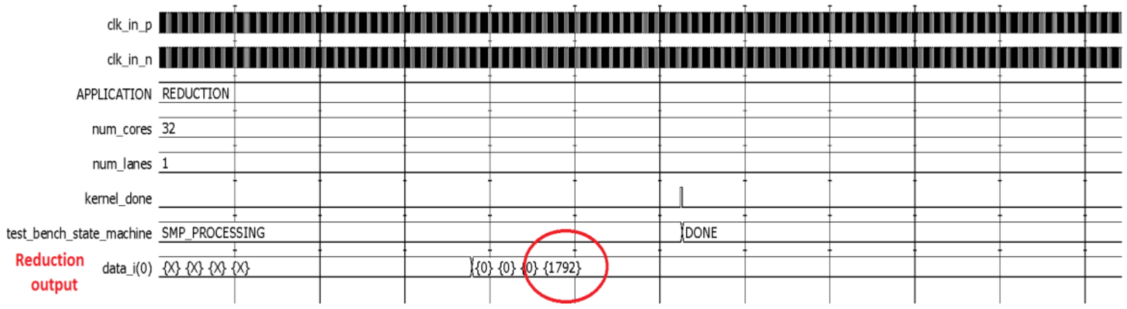


Figure 25: Reduction result

4.4 Chapter Summary

In this chapter, the software flow for executing a kernel on the soft GPGPU was described. The methodology for testing the Decode and Read-Write stages was elaborated and their results were presented. We presented the validation flow and described the benchmark suite. The chapter was concluded by presenting simulation results for two benchmarks.

CHAPTER 5

EXPERIMENTAL RESULTS

In this chapter, we describe the preliminary experiments conducted post-validation of the soft GPGPU. The experiments were mainly focused on evaluating the scalability of the platform in terms of the number of scalar processors (cores) as well as the number of streaming multiprocessors (SMs). The effects of scaling on area utilization are also investigated.

5.1 Performance evaluation

The platform was benchmarked against a MicroBlaze soft processor running on a Xilinx Virtex-6 ML605 evaluation board. ModelSim simulations were used to evaluate benchmarks on the soft GPGPU platform. The design was place and routed on the Virtex-6 device, and the post-PAR clock frequency along with simulation cycle counts were used to calculate the execution times. A software timer was used to time the MicroBlaze executions. Both platforms were operating at the same frequency of 100 MHz. For evaluating performance, two types of experiments were conducted—architecture scaling and application scaling as evaluated in the following sections.

5.1.1 Evaluating architecture scalability

A set of experiments were conducted to vary the number of cores within a single SM as 8, 16 and 32. Varying the number of cores effectively varies the number of threads in a row that can be executed in parallel. Recalling from chapter 3, GPU threads within a warp are scheduled as warp rows. This restricts the row width possibilities to 8, 16 or 32, as the product of row width and the number of rows (4, 2 or 1 respectively) must be a factor of 32.

Table 4 shows the cycle counts of the five benchmarks for a problem size as indicated. Ideally in a multicore system, as the number of cores is increased from 8 to 32, the expected performance improvement is 4x. The soft GPGPU shows an average speedup of 1.8x over the five benchmarks.

Cores	**Autocor 256**	**Bitonic 256**	**MatrixMul 256x256**	**Reduction 256**	**Transpose 256x256**	**Freq (MHz)**
8	2641050	952327	1247560898	65577	6207154	100
16	1832976	607695	876982560	46346	4752104	100
32	1441858	476820	693799691	37188	4026984	100

Table 4: Cycle counts comparison

Figure 26 shows the speedup graph normalized with respect to 8 cores. One common limitation to cycle speedup for all the benchmarks in our architecture is the scatter-gather memory instruction. Scatter-gather operations are most effective when the burst data is written and read in parallel. This requires the memory to be split up into multiple banks, such that consecutive memory addresses fall into consecutive banks. CUDA kernels are written in a way such that for most data-parallel applications, neighboring threads access consecutive memory locations. This allows threads to read data in parallel from consecutive memory banks. However, this demands the architecture to have sophisticated control mechanism to effectively map memory addresses to appropriate memory banks. For control flow intensive applications where the burst data is not sequential, this mapping must be done without significant overhead. The control logic becomes even more challenging to detect if multiple threads are pointing to the same address. For the sake of architectural simplicity, this feature was not included in our first soft GPGPU prototype and will be addressed in the future. The matrix benchmarks pay a slightly larger penalty for memory bandwidth limitations due to more number of scatter-gather operations. MatrixMul has a better performance than Transpose, as the former has higher arithmetic density and hence amortizes the bandwidth limitation to a certain extent.

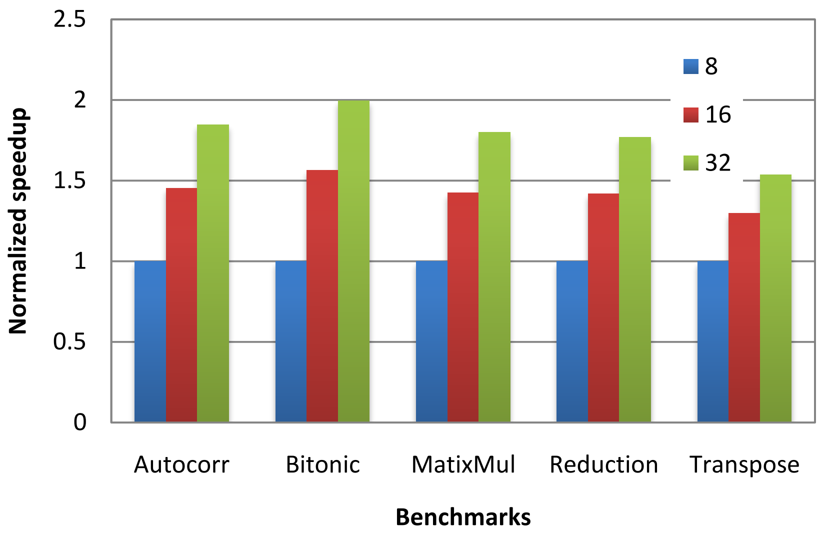


Figure 26: Performance scaling over 8, 16 32 cores

Figure 27 shows the calculated speedups against MicroBlaze for a varying number of cores. Application speedups range from 10x-30x with an average speedup close to 13x for 8 cores, 19x for 16 cores, and 25x for 32 cores. MatrixMul and Reduction being highly data parallel show the largest speedups. Reduction is a simple benchmark with a highly symmetric data flow graph consisting of multiple iterations. The number of array elements in the benchmark is halved with each iteration, progressively leading to smaller number of scheduled warps. Considering the array size to be a multiple of 32 (the warp size), all active threads remain tightly packed within a warp in every iteration, thus fully utilizing the warp at all times. In Bitonic, the sorting network consists of a fixed number of swapping operations that are performed at every stage. Though the warp divergence increases with increased number of parallel threads, the divergence cost seems to be amortized by performing more swapping operations in parallel. Transpose shows less speedup due to low arithmetic intensity and the memory bandwidth limitation.

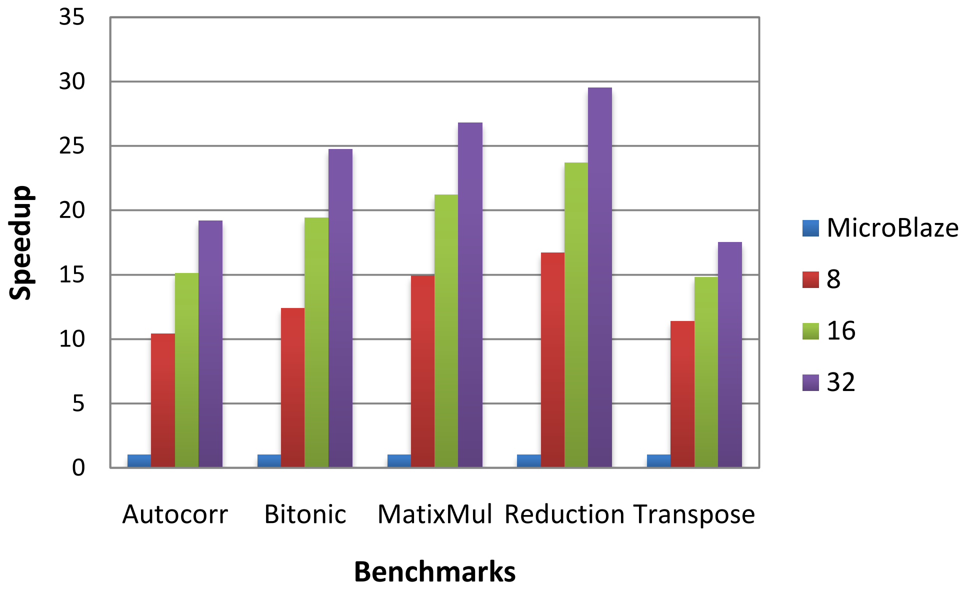


Figure 27: Speedup vs. MicroBlaze for variable cores

Another approach to explore the scalability of the architecture is by varying the number of SMs. This experiment was performed for MatrixMul and Transpose as these kernels can be split across multiple blocks. The block scheduler logic was modified to equally distribute thread blocks to 2 SMs, thus reducing the workload of each SM to half as before. Figure 28 shows the speedup for 1-SM and 2-SM configuration for the MatrixMul and Transpose.

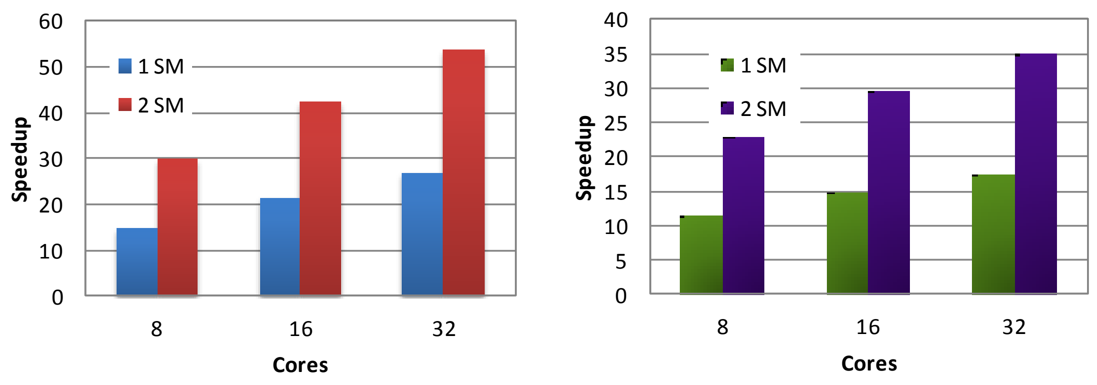


Figure 28: Speedup vs. MicroBlaze for variable SMs

(Left) MatrixMul; (Right) Transpose

5.1.2 Evaluating application scalability

Experiments were conducted to observe the performance of the soft GPGPU in comparison to MicroBlaze for varying problem sizes of each benchmark. The speedup results are shown in Figure 29.

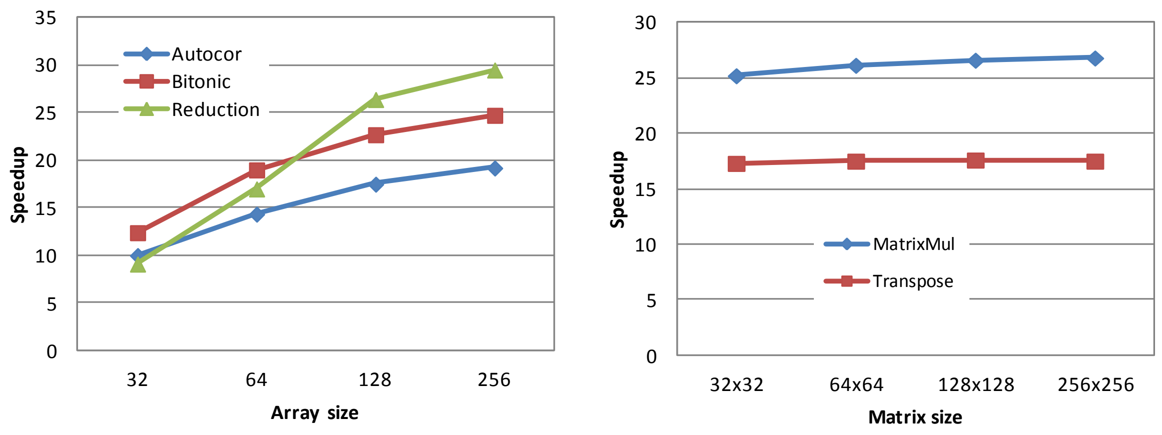


Figure 29: Speedup vs. Microblaze for varying problem size

Due to its regular kernel structure, Reduction reaps the steepest performance benefits of up to 30x as the size of the array becomes large. With increasing array size, performance increases gradually for both Autocor and Bitonic up to certain point and then begins to taper off. This can be attributed to the accumulation of the warp divergence penalty over the execution time of larger arrays, amortizing the parallel processing benefits. MatrixMul shows a reasonable speedup of about 25x, with Transpose showing tan average speedup of 17x. Both benchmarks have an almost flat speedup curve in accordance with the memory bandwidth limitation as addressed in section 5.1.1.

5.2 Area evaluation

The soft GPGPU design with 1 SM, 8 cores was synthesized, mapped, and successfully placed and routed on a Virtex-6 VLX240T device meeting all timing constraints. The post-PAR device utilization and maximum operating frequency are annotated in Table 5 .

Design characteristic	**1 SM / 8 cores per SM**
Logic used (LUTs)	63894 / 150720
Registers used (Flip Flops)	89392 / 301440
Multipliers used (DSP48E1s)	137 / 768
Block RAMs (RAMB36E1)	114 / 416
Maximum clock frequency (MHz)	100.05
Critical path	The scalar processor in the Execute stage

Table 5: Post-PAR utilization and timing results

The architecture takes advantage of the built-in multiplier blocks and BRAMs for performing computations and storing on-chip data. The critical path was found to be the module that performs integer addition subtraction within the scalar processor. As a target frequency of 100 MHz was achieved, this block was not optimized further.

In order to better understand the breakdown of area utilization, Xilinx PlanAhead tool [44] was used to gather utilization results of some of the blocks that consume relatively larger area (Table 6). The Read stage contains logic for reading three source operands in parallel in addition to the various register file and memory controllers, justifying the high resource utilization. The Write stage only consists of the register and memory controllers. The scalar processors in the Execute stage have dedicated compute units for supporting different types of arithmetic and logical instructions. Instantiating the scalar processor eight times duplicates logic reflecting the 34% LUT utilization. The stack memory (66 bits wide, 32 locations deep) used to handle divergence for each warp consumes 586 LUTs. Thus, for 24 warps 14064 LUTs are consumed.

Stage / Block	**LUT usage**	**Percent utilization**
Read	15290	24%
Execute	21499	34%
Write	6607	10%
Warp stack (24 warps)	14064	22%
Other	6524	10%
Total	**63984**	**100%**

Table 6: Area utilization breakdown

Additional results were gathered to study the effects of architecture scaling on area. Table 7 shows the post-synthesis device utilization statistics for a variable number of cores and SMs. The results are plotted in Figure 30 in order to examine the trend. The increase in the number of cores proportionally scales up the bit width of all the associated signals in the design, thereby reflecting near perfect linear scaling on LUTs and registers. The memory usage scaling presents a more interesting trend. The BRAM usage increases by approximately 25% from 8 to 16 cores and 24% from 16 to 32 cores. As discussed in section 3.2, register files are striped into memory banks with the number of banks equivalent to the number of cores in an SM. As the number of cores increases, the number of banks also increases, but with subsequent reduction in the size of each bank.

This is done to ensure that the total memory size remains constant. The reduction in the memory bank size might lead to their inefficient mapping onto the on-chip BRAMs—thus leading to higher BRAM utilization for more number of cores.

Architecture Configuration	**LUTs**	**Registers**	**Memory usage (BRAMs*)**
1 SM / 8 cores	60771	89024	79
1 SM / 16 cores	95292	126396	99
1 SM / 32 cores	196861	200055	123
2 SM / 8 cores	183068	338681	150

Table 7: Area for variable cores / SMs

***** Block RAMs are fundamentally 36 Kbits in size. Each block can also be used as two independent 18 Kb blocks.

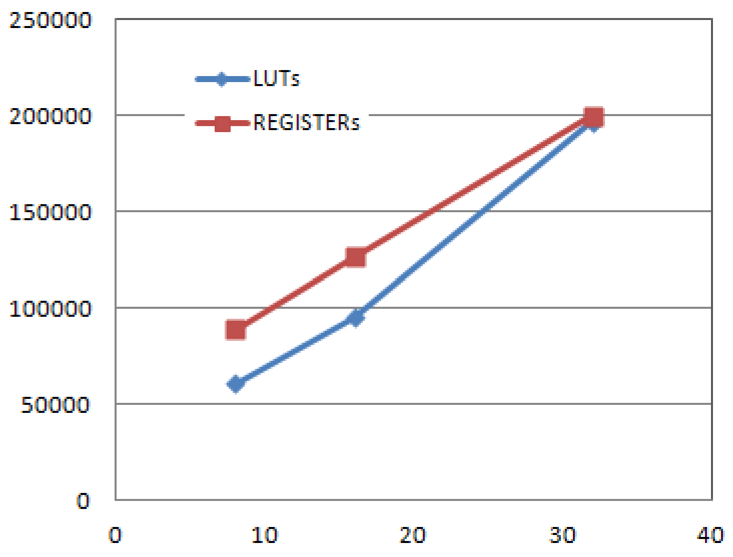


Figure 30: Variation trend for LUTs and Registers

5.3 Chapter Summary

In the beginning, the CUDA applications used for benchmarking the soft GPGPU platform were described. We analyzed the performance of our platform in comparison with a MicroBlaze soft processor for a varying number of cores/SMs and varying problem sizes. Speedups of up to 30x for single SM and up to 53x for two SMs were observed vs. MicroBlaze. To conclude the chapter, resource utilization for base system configuration is analyzed, with additional results to enunciate the effects of architecture scaling on area consumption. Next chapter concludes the thesis and provides future directions.

CHAPTER 6

CONCLUSION AND FUTURE WORK

In this thesis, efforts have been directed to implement a fully-functional, CUDA compatible, scalable soft GPGPU architecture targeting FPGAs. This document has outlined a systematic approach for testing and validation of the prototype soft GPGPU based on the Nvidia G80 architecture. A simulation based approach was adopted for testing and validating the system. Individual design blocks were subjected to functional RTL verification using VHDL testbenches and simulation tools. The novel design aspect of GPUs as opposed to standard microprocessors or even soft vector processors is the ability to handle thread divergence and barrier synchronization in hardware. Special care was taken to verify the correct synchronization and control flow behavior of the soft GPGPU. The system was integrated from scratch and validated using rigorous simulation for a set of five benchmarks directly compiled from CUDA to binary. The varied characteristics of the benchmarks allowed us to fairly evaluate the architecture. The binary was executed on the soft GPGPU without any further modifications.

Post validation of the base system (1 SM/8 cores), effort was directed towards augmenting the design for architectural scalability. The architecture was successfully enhanced to enable scaling the number of cores in the design as 8, 16 and 32. In addition, the design was also amended to enable scaling the number of streaming multiprocessors—a characteristic indigenous to Nvidia GPUs. A wide variety of experiments were conducted to evaluate the performance and area benefits of the soft GPGPU against a fully optimized MicroBlaze soft processor for a variable number of cores and SMs. Experimental results suggested speedups of up to 30x for highly parallel benchmarks like matrix multiplication and up to 24x for control flow intensive benchmarks like bitonic sort. Doubling the number of SMs resulted in a direct 2x performance improvement for matrix multiplication and transpose benchmarks with speedups up to 53x and 35x respectively. Area of the base soft GPGPU design was found to be 10x larger as compared to the MicroBlaze as most of the resources of the prototype architecture were spent towards ensuring correct CUDA functionality.

As with any prototype design, optimization would be the primary undertaking in the future. We also plan to improve out architecture by supporting off-chip memory access for global memory, multiple memory banks for efficient scatter-gather operations and implementing dynamic thread scheduling to reap true benefits of multithreading. We hope that the designed infrastructure sets the cornerstone for exploring an altogether new design space by facilitating rapid architectural tradeoffs and a wide variety of experiments in the future.

APPENDIX

MISCELLANEOUS DEBUGGING ISSUES

Simulink [28], developed by MathWorks, is a commercial tool that can be used to model design elements. It consists of the Xilinx blockset library that contains a set of customizable blocks for DSP, memory, arithmetic operations etc. In the soft GPGPU architecture, Simulink models are pervasively used to design larger and more complicated modules. The blocks using Simulink models include the warp unit, register files, scalar processors etc. Once a module is designed in Simulink using the inbuilt design blocks, it can be readily synthesized using the Xilinx System Generator tool. However, there were prevalent issues with simulating these modules within the Xilinx environment. This thesis involved debugging these issues and developing a systematic step-by-step procedure to import Simulink blocks and simulate them correctly.

Consider the scenario where two modules are modeled using Simulink and synthesized using the Xilinx System Generator (XSG). Let us assume that both modules use an adder as a sub-module with different bit-widths. The adder synthesized by the system generator within both the modules has generic bit widths, but the same name *xladdsub*. The *xladdsub* entity itself uses an instantiated adder core (with a unique name) to perform the addition. The core name is one of the generic inputs for *xladdsub* in addition to the bit-widths. For the correct operation of the *xladdsub* entity, this generic core name input must exactly match the adder core name instantiated in the entity. However, the *xladdsub* entity generated within each of the modules has a different core name. Thus, when an instance of *xladdsub* is declared in any of the modules, it is important for the generic input core name and the instantiated core names to match. This match would occur only if the *xladdsub* declarations in the two modules are linked to their own respective definitions. Considering that all the generated modules including the two versions of *xladdsub* are placed in a common “work” library, there is no way of differentiating between the two instances. This leads to mismatched linking between the *xladdsub* entity declarations for the two modules and their definitions, leading to undefined outputs during simulation.

A naïve solution is to manually rename the *xladdsub* entity declarations and definitions in each top level module with different names, for e.g. *xladdsub1* and *xladdsub2*. This differentiates the two versions of the *xladdsub* entity and makes sure each module finds its own version. However, this approach is cumbersome for large and complicated designs. A more systematic solution would be to make a separate library for each module that uses Simulink blocks. This ensures that all the different versions of overlapping entities like *xladdsub* are encapsulated into different libraries and there is no collision amongst them. A step-by-step procedure is illustrated below:

1) Open MATLAB 7.10.0.
2) Navigate to the directory containing the *.ngc* netlist folder corresponding to the top level module generated by XSG.
3) Run the following command in the command window:

   xlSwitchLibrary ('ngc_netlist', 'work', ‘user_defined_library’)

   This replaces all the references to the work library in the module file to the ‘*user_defined_library*, which will be created in the steps to follow.

4) Go to the libraries tab in ISE and create a new library with the same name as used in step 3 *(‘user_defined_library’)*.

5) Add the module to this library.

6) Now that the module is in the *user_defined_library* and not in the default *work* library, add library path to all other files referencing it.

   e.g. *library user_defined_library*.

7) Run simulation without any conflicts.

BIBLIOGRAPHY

[1] D. Sheldon, R. Kumar, R. Lysecky, F. Vahid, D. Tullsen, "Application-specific customization of parameterized FPGA soft-core processors," in *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design (ICCAD '06)*, ACM, New York, NY, USA, 261-268.

[2] P. Yiannacouras, J.G. Steffan, J. Rose. "Application-specific customization of soft processor microarchitecture", in *International Symposium on Field-Programmable Gate Arrays (FPGA)*, February 2006.

[3] E. Lindholm, J. Nickolls, S. Oberman, J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," Micro, IEEE, vol.28, no.2, pp.39-55, March-April 2008.

[4] Nvidia Corporation, Nvidia CUDA Programming guide, Version 2.3.1, August 2009.

[5] A. Gupta, "Formal hardware verification methods: A survey", in *Formal Methods in System Design*, 1, 2/3 (Oct. 1992), 151–238.

[6] C. Kern and M. R. Greenstreet, "Formal verification in hardware design: A survey," in *Transactions on Design Automation of Electronic Systems*, 4:123–193, 1999.

[7] http://en.wikipedia.org/wiki/Field-programmable_gate_array

[8] K.S. Pereira, "Characterization of FPGA-based High Performance Computers," Master's Thesis, Virginia Polytechnic Institute and State University, Virginia, 2011.

[9] J. Fowers, G. Brown, P. Cooke, G. Stitt, "A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications," in *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays (FPGA '12*), ACM, New York, NY, USA, 47-56.

[10] X. Tian, K. Benkrid, "High-performance quasi-Monte Carlo financial simulation: FPGA vs. GPP vs. GPU," in *ACM Transactions on Reconfigurable Technology and Systems*, 3, 4, Article 26, November 2010, 22 pages.

[11] J. Cong, Bin Liu, S. Neuendorffer, J. Noguera, K. Vissers, Z. Zhang, "High-Level Synthesis for FPGAs: From Prototyping to Deployment," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol.30, no.4, pp.473-491, April 2011.

[12] R Suda, T. Aoki, S. Hirasawa, A. Nukada, H. Honda, S. Matsuoka, “Aspects of GPU for general purpose high performance computing,” in *Proceedings of the 2009 Asia and South Pacific Design Automation Conference (ASP-DAC '09)*, IEEE Press, Piscataway, NJ, USA, 216-223.

[13] T. Dokken, T.R. Hagen, J.M. Hjelmervik, “The GPU as a high performance computational resource,” in *Proceedings of the 21st spring conference on Computer graphics(SCCG '05)*, ACM, New York, NY, USA, 21-26.

[14] Chi-Hung Chi, Siu-Chung Lau, “Reducing data access penalty using intelligent opcode-driven cache prefetching,” in *IEEE International Conference on Computer Design: VLSI in Computers and Processors, ICCD '95,* 1995, vol., no., pp.512-517, 2-4 Oct 1995.

[15] J.B. Chen, M.D. Smith, C. Young, N. Gloy, “An Analysis of Dynamic Branch Prediction Schemes on System Workloads,” in *23rd Annual International Symposium on Computer Architecture*, 1996, vol., no., pp. 12, 22-24 May 1996.

[16] E.Z. Zhang, Y. Jiang, Z. Guo, K. Tian, X. Shen, “On-the-fly elimination of dynamic irregularities for GPU computing,” in *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems (ASPLOS '11)*, ACM, New York, NY, USA, 369-380.

[17] S. Collange, M. Daumas, D. Defour, and D. Parello, “Barra: A Parallel Functional Simulator for GPGPU,”in *IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2010 , vol., no., pp.351-360, 17-19 Aug. 2010.

[18] http://www.nvidia.com/object/tesla_computing_solutions.html.

[19] G. Ruetsch, B. Oster, “Getting Started with CUDA*,” nVision 08: The world of visual computing*, 2008.

[20] I. Lebedev, C. Shaoyi, A. Doupnik, J. Martin, C. Fletcher, D. Burke, L. Mingjie, J. Wawrzynek, “MARC: A Many-Core Approach to Reconfigurable Computing,” in *International Conference on Reconfigurable Computing and FPGAs (ReConFig), 2010*, vol., no., pp.7-12, 13-15 Dec. 2010.

[21] C. Fletcher, I. Lebedev, N. Asadi, D. Burke, J. Wawrzynek, “Bridging the GPGPU-FPGA Efficiency Gap,” in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays (FPGA '11),* Feb 27 - Mar 1, 2011.

[22] J. Kingyens and J.G Steffan, “A GPU-inspired soft processor for high-throughput acceleration,” in *IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010*, vol., no., pp.1-8, 19-23 April 2010.

[23] A. Papakonstantinou, K. Gururaj, J. Stratton, D. Chen, J. Cong and W. Hwu, “FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs,” in *Application Specific Processors, 2009*, SASP '09, IEEE 7th Symposium on , vol., no., pp.35-42, 27-28 July 2009.

[24] Altera, “Implementing FPGA Design with the OpenCL Standard,” white paper, November 2011.

[25] Xilinx Inc., “MicroBlaze Processor Reference Guide – Embedded Development Kit EDK 14.2”, July 2012.

[26] Nvidia Corporation, PTX: Parallel Thread Execution ISA, Version 2.3, March 2011.

[27] Xilinx Inc., LogiCORE IP Block Memory Generator v4.3, September 2010.

[28] http://www.mathworks.com/products/simulink/.

[29] Xilinx Inc., “System Generator for DSP – User Guide”, March 2011.

[30] http://en.wikipedia.org/wiki/CUDA

[31] University of California, Berkeley, GPGPU-Sim 3.x Manual, http://gpgpu-sim.ece.ubc.ca/Main_Page.

[32] Nvidia Corporation, cuobjdump - Application Note, DA-05536-001_v03, January 2011.

[33] http://developer.nvidia.com/cuda-toolkit-23-downloads.

[34] ModelSim SE, http://model.com/node/16.

[35] Wladimir J Van der Laan, https://github.com/laanwj/decuda.

[36] S. Collange, M. Daumas, D. Defour, and D. Parello, “Barra: A Parallel Functional Simulator for GPGPU,”in *IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2010 , vol., no., pp.351-360, 17-19 Aug. 2010.

[37] A. Bakhoda, G. Yuan, W. W. L. Fung, H. Wong and T. M. Aamodt, “Analyzing cuda workloads using a detailed GPU simulator,” in *IEEE ISPASS*, April 2009.

[38] https://svn.ece.lsu.edu/svn/gp/cuda/matrix-mult/mm-gt200.sass

[39] http://www.cs.duke.edu/courses/fall08/cps196.1/Pthreads/bitonic.c

[40] http://ercbench.ece.wisc.edu/

[41] www.cse.nd.edu/courses/cse60881/www/lectures/logsum.pdf

[42] http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/bitonicen.htm

[43] http://www2.tech.purdue.edu/cgt/Courses/tech621/lectures/TECH621GPGPU-09-CUDA%20Parallel%20Reduction.pdf

[44] Xilinx Inc., "PlanAhead User Guide", December 2009